UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft

Performance Modeling and Analysis in High-Performance Reconfigurable Computing

Dissertation

von
Tobias Schumacher

Schriftliche Arbeit zur Erlangung des Grades
eines Doktors der Naturwissenschaften

Fakultät für Elektrotechnik, Informatik und Mathematik
der Universität Paderborn

Acknowledgements

I would like to thank:

- My advisor Prof. Marco Platzner for his support and encouragement. I benefited a lot from his experience and always got valuable feedback when discussing my ideas and thoughts.
- Jun.-Prof. Dr. André Brinkmann for agreeing to review this thesis and to serve as a co-examiner for my dissertation.
- Dr. Christian Plessl for his advice during my research, many fruitful discussions and for agreeing to serve as a member of the oral examination commission.
- Prof. Dr. Sybille Hellebrand and Prof. Dr. Franz Rammig for serving as members of the oral examination commission.
- My colleagues at PC^2 and the Computer Engineering group for providing me with an excellent research environment, a great atmosphere, technical support and invaluable discussions.
- Kerstin Voß, Michelle Shuttleworth and Steffi Funke for reviewing and improving the grammar and spelling of this thesis.
- My parents, my brothers and my friends for continuous encouragement and support.

Abstract

Reconfigurable computing has received a high level of attention during the last years. Scientists presented accelerators for different algorithm classes gaining speedups of several orders of magnitude. Major supercomputer vendors came out with high-performance computers that tightly connect reconfigurable devices to the CPUs and/or to the memory subsystem. One of the major focuses of recent research is put on the programmability of these reconfigurable high-performance computers. Despite great research results in this topic, there are still several challenges which make the development process of reconfigurable accelerators a time consuming and error-prone process.

One of the main issues in this area is the question whether reconfigurable computing is even able to generate a benefit for specific applications. Since the design of reconfigurable accelerators typically is a very time consuming process, it is mandatory to estimate the potential of this technology before actually implementing the accelerator. For this purpose, modeling techniques are required. Existing modeling approaches are often restricted to a static architecture model and provide methods for specifying algorithms to be executed on those specific architecture models. These techniques are not well-suited when considering reconfigurable computing, since in these cases the concrete architecture is typically generated explicitly for the specific algorithms.

In order to analyze the performance potential of the generated accelerator model, a deep knowledge of the underlying architecture is required. This includes especially the time necessary for data transfers between CPU and accelerator as well as the time needed for memory access. Many tools exist for measuring low level performance values on commodity CPUs, but no corresponding tools are available to measure such values for reconfigurable hardware.

The design and implementation of reconfigurable accelerators lead to the demand for a development framework that supports the designer in implementing and testing the

modeled design. Simulating a complete design typically requires a large amount of time, so the development framework should support in-system performance monitoring. Another key issue is the portability of the framework and the resulting accelerators.

This thesis introduces a novel approach to meet these requirements. It introduces a modeling technique which supports algorithms targeted at commodity CPUs as well as reconfigurable accelerators. A key point of the modeling technique is that it does not assume a static architecture model, but allows for specifying the architecture model along with the execution model of the algorithms to be implemented. Additionally, the modeling approach does not only focus on the execution time of arithmetic operations performed, but also on the time needed for data transfers.

The modeling approach is supported by the IMORC architectural template which eases the implementation of the modeled accelerator. The architectural template assumes accelerators to be implemented as a set of communicating cores. Each core may reside in its own clock domain and communicate to others using an on-chip network. A key feature is that the network allows control structures and datapaths to be implemented completely independently from each other. Integrated performance counters support the debugging and the in-system performance analysis of the final accelerator.

In order to further support the modeling phase, an architecture characterization framework based on the architectural template is introduced. This framework allows to measure the communication bandwidth between CPU and reconfigurable hardware as well as between reconfigurable hardware and different kinds of memory in detail. It supports different kinds of communication schemes and can also generate contention on the target memory by accessing it concurrently using multiple cores.

The introduced approach is finally evaluated by demonstrating three case studies out of different problem domains. These case studies show that the presented approach greatly helps in analyzing algorithms concerning their acceleration potential on reconfigurable hardware and in implementing and optimizing the final accelerators. The accelerators are implemented using only a small amount of hand written VHDL code. Most functionalities are realized using the features of the IMORC architectural template. The integrated load sensors helped significantly to identify bugs and performance bottlenecks during the design phase. Those would have been hard to find using only simulation techniques.

Zusammenfassung

Rekonfigurierbares Rechnen hat in den vergangenen Jahren einen hohen Grad an Aufmerksamkeit erhalten. Wissenschaftler zeigten Beschleuniger für unterschiedliche Klassen von Algorithmen, die Geschwindigkeitssteigerungen von mehreren Größenordnungen erreichten. Die Bedeutung von rekonfigurierbarem Rechnen lässt sich auch daran erkennen, dass namhafte Anbieter von Hochleistungsrechnern Systeme mit rekonfigurierbarer Hardware, die eng an die CPU und das Speichersystem gekoppelt ist, entwickelten. Ein besonderer Schwerpunkt der Forschung liegt in der Programmierbarkeit solcher rekonfigurierbarer Rechner. Trotz nennenswerten Ergebnissen auf diesem Gebiet existieren allerdings weiterhin einige Herausforderungen, die den Entwicklungsprozess rekonfigurierbarer Beschleuniger zeitaufwändig und fehleranfällig machen.

Insbesondere stellt sich die Frage, welchen Nutzen eine Realisierung anhand von rekonfigurierbarem Rechnen für eine bestimmte Applikation bietet. Der zeitaufwändige Entwicklungsprozess erfordert eine genaue Abschätzung der Nutzbarkeit von rekonfigurierbaren Beschleunigern bereits vor der tatsächlichen Implementierung. Hierzu werden Modellierungsmethoden benötigt. Bestehende Modellierungstechniken basieren häufig auf einem festen Architekturmodell und stellen Methoden zur Verfügung, um Algorithmen für die Ausführung auf dieser Architektur zu spezifieren. Solche Methoden eignen sich schlecht für die Modellierung von rekonfigurierbarer Hardware, da in diesem Fall die Architektur nicht vorgegeben ist, sondern explizit für die zu implementierenden Algorithmen entworfen wird.

Um das Geschwindigkeitspotential eines Beschleunigermodells zu analysieren, wird eine tiefgreifende Kenntnis der zugrundeliegenden Architektur benötigt. Dies beinhaltet vor allem die Zeiten, welche für die Datentransfers zwischen CPU und Beschleuniger sowie für Speicherzugriffe benötigt werden. Es existieren viele Hilfsprogramme, um solche Parameter für typische CPUs zu messen, allerdings mangelt es an äquivalenten Lösungen um entsprechende Werte für rekonfigurierbare Hardware zu bestimmen.

Das Design und die Implementierung rekonfigurierbarer Hardware erzeugen weiterhin

einen Bedarf an Frameworks, die dem Entwickler bei der Implementierung und dem Testen der modellierten Beschleuniger Unterstützung bieten. Die Simulation vollständiger Schaltungen ist typischerweise sehr zeitaufwändig. Dementsprechend sollte solch ein Framework Unterstützung für die Geschwindigkeitsanalyse im laufenden System bereitstellen. Ein weiterer wichtiger Punkt ist die Portabilität eines solchen Frameworks und der darauf basierenden Beschleuniger.

In dieser Arbeit wird ein neuer Ansatz zur Lösung dieser Anforderungen vorgestellt. Die Arbeit führt eine Modellierungsmethode ein, die die Spezifikation von Algorithmen unterstützt, welche auf CPUs oder auf rekonfigurierbarer Hardware ausgeführt werden sollen. Weiterhin betrachtet der vorgestellte Modellierungsanzatz nicht nur die Ausführungszeit arithmetischer Operationen. Insbesondere wird auch die benötigte Zeit für Datentransfers berücksichtigt.

Der Modellierungsansatz wird durch das IMORC-Architekturtemplate unterstützt, welches die Implementierung der modellierten Beschleuniger vereinfacht. Das Architekturtemplate basiert auf der Annahme, daß Beschleuniger aus einer Menge an kommunizierenden Kernen bestehen. Jeder Kern kann in einer eigenen Taktdomäne arbeiten und über ein On-Chip-Netzwerk mit anderen Kernen kommunizieren. Ein besonderes Merkmal des Netzwerkes ist, daß es die Möglichkeit bietet, Kontrollstrukturen unabhängig von den Datenpfaden zu implementieren. Weiterhin stellt es integrierte Lastsensoren zur Verfügung, die eine Performanceanalyse und das Debugging des entwickelten Beschleunigers zur Laufzeit im System ermöglichen.

Um die Modellierungsphase weiter zu unterstützen, wird darüber hinaus ein Framework zur Charakterisierung der Zielplattform vorgestellt, welches auf dem IMORC-Architekturtemplate basiert. Dieses Framework ermöglicht die detailierte Messung der zur Verfügung stehenden Bandbreite zwischen CPU und rekonfigurierbarer Hardware sowie zwischen rekonfigurierbarer Hardware und den verschiedenen Arten an verfügbarem Speicher. Es unterstützt unterschiedliche Zugriffsmuster. Darüber hinaus kann es konkurierende Zugriffe auf den Zielspeicher generieren, indem mehrere Kerne gleichzeitig auf den Speicher zugreifen.

Das vorgestellte Verfahren wird anhand dreier Fallstudien aus verschiedenen Domänen evaluiert. Diese Fallstudien zeigen zum einen, daß das vorgestellte Verfahren bei der Eignungsanalyse von Algorithmen für eine Umsetzung mit rekonfigurierbarer Hardware hilreich ist. Zum anderen wird deutlich, daß es die Implementierung und Optimierung der enwickelten Beschleuniger stark vereinfacht. Die Implementierung der Beschleuniger erforderte nur wenig handgeschriebenen VHDL-Code. Der Großteil der Funktionalitäten konnte durch die Funktionen des IMORC-Architekturtemplates realisiert werden. Die integrierten Lastsensoren erleichterten das Aufspüren von Fehlern und Leistungsengpässen erheblich. Diese Informationen wären mit reinen Simulationstechniken nur sehr schwer zu identifizieren gewesen.

Contents

1 Introduction

Since many years academic research has studied the use of application-specific coprocessors based on field-programmable gate arrays (FPGAs) to accelerate high-performance computing (HPC) applications. Several publications discuss the implementation of accelerators for different kinds of applications.

While early work performed in this area usually used workstations equipped with traditional PCI or PCIe attached FPGA accelerator boards, in the recent years major supercomputer vendors started offering servers with integrated reconfigurable accelerators tightly connected to the system's high-performance interconnect. Such integrated solutions are able to harness one of the major issues of traditional accelerator boards — the communication bandwidth between host processor/memory and the accelerator.

However, designing an accelerator and optimizing its performance still remains a difficult and time consuming task requiring significant hardware design expertise. Several different approaches target at simplifying this implementation task, such as visual design tools and high-level language compilers.

This thesis aims at guiding the accelerator design process with a model-based approach that enables performance optimization throughout the design flow. The approach uses different modeling techniques to estimate the effects of different architectural decisions, monitoring actual performance values in real systems and runtime-optimization.

1.1 Motivation

Many tools are available in traditional high-performance computing that support developers with optimizing applications for specific platforms. Profiling tools exist for identifying the most computation intensive parts of an application. While from the applications' perspective those parts are the best candidates for acceleration, this does not imply that reconfigurable hardware is a suitable technology for accelerating these parts. Implemented

on FPGAs, lots of applications might achieve speedups of several orders of magnitude, others might even perform worse on FPGAs when compared to optimized CPU implementations. Even when algorithms are generally suitable for FPGA acceleration, the performance of the complete application including the accelerator often depends on system specific parameters, such as the communication performance between the FPGA and the memory location where the problem resides.

Although with the wide variety of tools supporting the design and implementation of reconfigurable accelerators, the design and performance optimization step still remains a difficult and time consuming task. Trial-and-error methods for generating reconfigurable accelerators cannot be esteemed as efficient. Modeling techniques are required to estimate the performance gains achievable by using reconfigurable accelerators in general and on the specific target platforms available before actually designing, implementing and optimizing the accelerator. Such modeling techniques should be applicable at a very early stage of the design phase, with little or no information of the final accelerator design available.

When this estimation shows that FPGA acceleration is feasible, the design process begins. Important design decisions based on the application and the target architecture influence the final performance of the accelerator and have to be taken carefully: Various storage locations are available for storing intermediate data, but all of them may have different performance characteristics that could influence the accelerator's performance; contention may occur when resources are shared among multiple execution units for example. Therefore, again modeling techniques are required to design an efficient accelerator.

Such a design approach needs a detailed knowledge of the target architecture. Parameters like the amount of memory available in different locations, the bandwidth available when accessing such memories and the communication bandwidth between different processing elements are especially interesting. Vendors usually provide such information; however these communication and memory access bandwidths usually are theoretical peak values that are not achievable in real applications. For getting detailed knowledge of the target platform, microbenchmarks have to be created.

Following the design phase, the accelerator is implemented. Several tools and languages try to ease the implementation phase, all providing different assets and drawbacks. The success of the implementation greatly depends on an adequate selection of used tools and methods. With a specific modeling approach taken for the design phase, ideally specific methods and tools are available supporting the implementation of designs resulting from these modeling techniques. Additionally, when the implementation is valid, supporting methods are needed for debugging and optimizing the implementation.

1.2 Contributions of this Thesis

The contribution of this thesis is a design and implementation flow for reconfigurable accelerators with a focus on high-performance computing. More precisely, the contributions are as follows:

- A modeling flow for reconfigurable accelerator design is introduced. The model consists of an architecture model describing the architecture of the target platform and an execution model describing the application. The execution model especially characterizes the communication behavior of the application. Mapped to the architecture model, the communication and execution time can be roughly estimated. In the design phase, this mapped execution model is refined for further detailing the FPGA accelerator in the architecture model, which can then be implemented on the actual target platform.

- Based on this modeling flow, this thesis introduces a novel application template supporting the implementation, optimization and debugging of the accelerator. The template is based on the Infrastructure for Performance Monitoring and Optimization of Reconfigurable Computers (IMORC), an infrastructure for implementing accelerators consisting of multiple communicating cores. Additional functionalities are available for implementing tasks often needed. The application template was designed to match the modeling approach presented in this thesis, enabling the developer to map the application model directly to the resources available. Additionally, the template provides methods for measuring the performance and finding bottlenecks in a running system, simplifying further performance optimizations.

- A benchmarking framework for accurately characterizing target platforms is presented. The framework can measure the bandwidth of communication channels from the FPGA to different memory locations and between the host CPU and cores on the FPGA. It does not only measure the peak values achievable, but can also simulate different communication schemes and contention on the target resource.

1.3 Thesis Outline

The thesis is organized as follows:

Chapter 2 *Background and Related Work* provides the background to this thesis and summarizes related work. It presents an introduction to HPC computing and FPGA computing and gives an overview of existing approaches for performance estimation, design and implementation of FPGA based accelerators.

Chapter 3 *Programming, Execution and Performance Model* discusses the modeling approaches used within this work. It first presents the programming model and methods for partitioning given problems into a set of tasks that communicate with each other. The remainder of this chapter discusses the analysis and optimization of the resulting task graphs for maximizing the overall performance.

Chapter 4 *The IMORC Architectural Template* introduces the IMORC Infrastructure for Performance Modeling and Optimization of Reconfigurable Computers. The first part discusses the architecture of the IMORC communication infrastructure and how the models presented in Chapter 3 can be mapped to the IMORC architectural template. Several generic cores are presented that implement subtasks often needed by accelerators. Additionally, as a basis for the case studies presented in the following chapters the XtremeData XD1000 platform and the corresponding IMORC interface cores are introduced. The second part presents the features IMORC provides for debugging and optimizing accelerators by gathering performance values from the system during runtime.

Chapter 5 *Architecture Characterization* discusses a set of benchmarks based on IMORC that are used for characterizing the communication performance of different kinds of memory on individual target architectures. These characterizations form the foundation of a concrete mapping of data to the individual memories for maximizing the accelerators performance. A detailed performance analysis for the XtremeData XD1000 is presented.

Chapter 6 *Experimental Evaluation* demonstrates some case studies based on the previous three chapters. It introduces some concrete applications from different domains that are analyzed using the modeling techniques presented in Chapter 3 and presents accelerator implementations for these algorithms based on these modeling techniques and the infrastructure presented in Chapter 4. The architecture characterization presented in Chapter 5 combined with the application model thereby support finding a good mapping of data to the memory resources available. This way, this chapter demonstrates the benefits achieved by the techniques introduced in the previous chapters quantitatively compares the accelerators to equivalent CPU implementations.

Chapter 7 *Conclusion and Outlook* finally summarizes the achieved contributions. It draws an overall conclusion and gives an outlook on possible future research directions in this area.

2 Background and Related Work

This chapter summarizes background information of this thesis and related work. First, a brief overview of current trends in the area of accelerated supercomputing is presented. In particular some of the current platforms with integrated FPGA accelerators are introduced. Second, an overview of related work in modeling FPGA accelerators and performance estimation techniques is given. Third, existing design methods and tools used for implementing, verifying and optimizing reconfigurable accelerators are discussed in regard to benefits and limitations of the different approaches.

2.1 Accelerated Supercomputing

Since many years acacemic research has studied different approaches for accelerating high-performance computing applications. Companies provide special vector processors, such as the ClearSpeed processor [10], which is able to provide high speedups for different classes of algorithms. Other trends are the utilization of commodity graphics processing units (GPU) for performing computations. While these coprocessors often provide very good results with low effort, their use still is restricted to several classes of algorithms. FPGA-based application specific coprocessors are an alternative to such vector processors, which often can provide good speedups for algorithms that do not perform well on such vector processors. FPGAs gain performance if many operations can be performed in parallel on a small set of data, for example by pipelining operations.

Clusters of workstations were equipped with reconfigurable hardware for accelerating computation intense kernels of applications. Examples are given by the Hydra [51] chess computer operated by the PAL group in Abu Dhabi, the Maxwell cluster operated by the Edinburgh Parallel Computing Centre (EPCC) [12] for the FPGA High Performance Computing Alliance [13], Novo-G operated by the NSF Center for High-Performance

Reconfigurable Computing (CHREC) [17] and the Axel cluster operated by the Imperial College London [71, 123]. Application specific accelerators were developed, such as [1], [78] and [126]. A major challenge in the accelerator design was the communication between processors and reconfigurable logic, especially in cases FPGA hardware is connected to clusters of workstations using PCI or PCIe.

To overcome such issues, in the recent years, major supercomputer vendors reduced these restrictions by providing servers with integrated reconfigurable accelerators, tightly connected to the main CPUs and memory. Examples are given by the Cray XD1 [11] system, the SRC MAP [24] processor available for the SRC-6 and SRC-7 and the SGI RASC [23] blade available for integration into the Altix line of systems. These three families of high-performance systems integrate a large number of processors using a proprietary interconnect into a NUMA-style (Non-Uniform Memory Access) shared memory system. The FPGAs in these systems are directly connected to this interconnect, allowing them to communicate with the main memory at a very high speed.

Another approach followed in the last years was to directly integrate the FPGA into a commodity CPU socket of standard workstations. Given a multi-processor mainboard, one or several CPU sockets are equipped with CPUs, the others with FPGA accelerators. Examples are given by the XtremeData XD1000i and XD2000i [128], which integrate the FPGA into an AMD Opteron socket. Since in Opteron based systems each CPU socket is connected to a unique bank of memory, the FPGA in these systems can directly access its own dedicated large area of DDR/DDR2 memory. Additionally, the accelerator's printed circuit board (PCB) provides a small amount of low latency SRAM, which is accessible by the FPGA. CPU and FPGA can communicate to each other using a 16 bit HyperTransport link at a rate of 800 MT/s, resulting in a theoretical peak bandwidth of 2×1.6 GB/s

The XtremeData XD2000F and the Nallatech FPGA-FSB [15] platform integrate an FPGA in a similar way into the Front Side Bus (FSB) of Intel Xeon based servers. All sockets are connected to the central memory controller using the FSB, thus sharing the same main memory. As a result, the FPGAs have not any exclusive access to large DDR/DDR2 SDRAM memories, but only to some smaller areas of SRAM available directly on the PCB.

2.2 Performance Modeling, Analysis and Optimization

A key point in the design of reconfigurable accelerators is the achievable overall performance. For traditional computing and HPC computing several research has been conducted to analyze the performance of applications running on a special kind of hardware.

2.2.1 Performance Analysis in High-Performance Computing

Lots of research has been conducted to analyze the performance of applications in the area of traditional high-performance computing. The objective of such work is to analyze the application's behavior on a certain computing platform and addresses different aspects. The approaches can be grouped into different classes:

Profiling

Profiling is a technique that can give an insight into the runtime behavior of an application using real workloads. Software can be instrumented for example by automatically inserting additional code for generating profiling data. Alternatively or additionally performance counters available in hardware can be used for collecting performance data. Profiling is often used in performance tuning of already existing applications. Frequently called methods of the application can be identified in order to concentrate on these critical points in the software. Additionally, bottlenecks introduced by the system can be identified by monitoring the resource usage of its components like network, memory and so on.

The knowledge of critical segments in the application and their runtime behavior, such as the demand for data access and communication or the number and type of operations performed, forms the basis to estimate the suitability of reconfigurable hardware for acceleration of these segments. An upper bound on the achievable speedup can for example be estimated by analyzing the data requirements and calculating the time needed for data transfers to the reconfigurable hardware. By Amdahl's law [33] this rough speedup estimation can be transformed into an upper bound on the speedup realizable for the complete application. Additionally, the extracted data can be used for parameterizing analytical models of the application.

A key issue in using profiling tools for gaining performance values is that the application is required to be already implemented and running on a concrete system. On the one hand, instrumenting the application may influence the runtime behavior of the application, resulting in potentially corrupted measurements. On the other hand, if only using hardware monitors to gather runtime performance values the amount of data that can be extracted is restricted to the performance sensors actually available.

The GNU profiler gprof [57] is an example for a popular profiling tool using instrumentation. The application to be profiled is compiled and linked using the GNU compiler suite with profiling enabled. During execution it generates a profile data file that can be analyzed using the gprof tool.

OProfile [21] is a well-known profiling tool for Linux-based systems. It consists of a kernel driver to access hardware performance counters, a pseudo-filesystem for communication with the userspace and an OProfile daemon that communicates with the kernel interface and stores traces to the disk and tools for later analyzing the collected data. Since the tool uses the internal performance counters for collecting data, no overhead is

introduced into the application's runtime itself. Only the system load is slightly increased due to the running OProfile daemon storing the collected data.

Intel's VTune [74] and the Sun Studio Performance Tools [119] are examples for other profiling tools, which are designed to gather performance information from applications running on single-processor workstations up to large scale shared memory systems or even MPI applications running on beowulf clusters. This property makes those tools very popular in the HPC community. Both tools use non-intrusive profiling, so no recompilation of existing code has to be performed.

Valgrind [26, 90] uses another approach for gathering such information. It consists of a core that provides a virtual processor and tools that are based on this core. The application to be profiled is disassembled into an intermediate representation (IR) which is then instrumented with analysis code. The code is executed on the virtual processor with the instrumentation code collecting the performance data as defined by the tool. However, since the application is executed on a virtual processor, performance values are only collected for the original code. For example, if monitoring the runtime of application segments, only the runtime for the original segment is collected, the overhead imposed by the instrumentation code is not added to this runtime. Additionally, several parameters of the virtual processor such as the cache may be specified by the user, simulating the execution on different architectures. These properties make Valgrind a very accurate tool, but the simulation of a virtual processor also reduces the execution performance.

Modeling

Modeling is another approach for gathering an insight into the runtime requirements of applications. Analytical models usually consist of an abstract model of the target architecture and a specification of the application. The execution of the application model on the architecture model is simulated and the runtime is calculated, usually depending on parameters like the input size.

Such models are frequently used in theoretical computer science for classifying algorithms by their asymptotic behaviour. Examples are given by the Random Access Machine (RAM) [43] and the Random Access Stored Program (RASP) [52], which are simplified models of the Harvard architecture and von Neumann architecture, respectively. Advanced features of modern processors, such as memory hierarchies and pipelining, are not considered by such models. For the evaluation of parallel algorithms and contention on shared memory resources, the Parallel Random Access Machine (PRAM) [56] was introduced, a simplified model of symmetric multiprocessing systems (SMP). The classification by the asymptotic behavior makes algorithms comparable. Figure 2.1 shows the principal architecture diagram of the PRAM.

However for estimating the runtime on real systems, these methods are usually not accurate. Especially the time needed for communication between processors and for

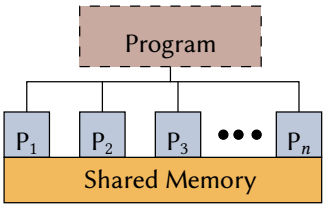


Figure 2.1: Diagram of the PRAM architecture

memory access is neglected. Resulting, applications using very fine-grained parallelism may perform great on the PRAM while a real implementation might even perform worse than a single-threaded implementation. The Queuing Shared Memory (QSM) [100] model was introduced to overcome this issue. It assumes a shared memory system, where each processor additionally contains a certain amount of local memory (cache, registers). Algorithms are modeled using three phases: 1) reads from shared memory, 2) writes to shared memory and 3) local operations. The phases are synchronized; as the reads or writes may be concurrent, cost functions are provided for such concurrent accesses.

All models presented above assume some kind of shared memory for interprocess communication. For also supporting systems that communicate using message passing, several other models were developed. The Bulk Synchronous Programming (BSP) [125] model assumes an application to be running on a system with multiple processors. In BSP, the application's execution is divided into three phases (cmp. Figure 2.2):

1. a computational phase, in which only local operations are performed,
2. a communication phase, in which the processors exchange messages and
3. a synchronization phase, where all processors synchronize using a barrier operation.

After the synchronization the processors continue with the next computational phase.

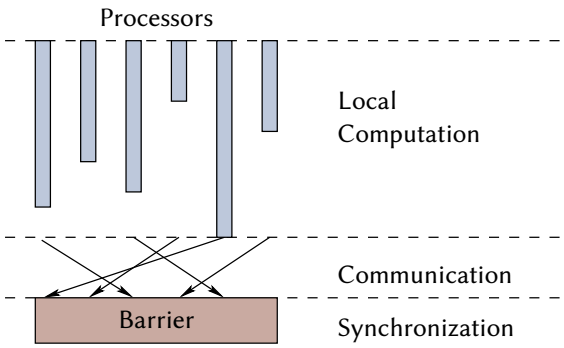


Figure 2.2: Execution diagram of the BSP model with six processors

The LogP [45] model predicts the communication performance of an application by assuming processors to communicate using point-to-point messages. The underlying architecture model implies a number of compute nodes that communicate using a network. The architecture is characterized by four parameters: *Latency* (L), *overhead* (o), *gap* (g) and the number of *Processors* (P). With the communication model of the application and these parameters, the overall communication performance of the application is predicted. A problem is that the parameters in this model are static, assuming only fixed sized messages to be transferred between the processors. LogGP [30] is an extension to this model which additionally comprises messages of different sizes. The Latency of Data Access (LDA) [109] is another modeling approach which supports shared memory systems as well as clusters of workstations. The architecture model consists of a number of processors that are connected to some kinds of local memory (registers, caches) and potentially some shared memory. Additionally, the processors may be connected by some kind of network. The application is modeled as a set of communicating tasks, which are mapped to the available processors in the next step. Tasks consist of local operations, memory accesses to the different kinds of memory and network access. Operations are classified due to their execution time, such as integer operations, floating point communication, access to L1/L2/L3-cache or shared memory and so on.

A different modeling approach is followed by the Kahn Process Network (KPN) introduced by Kahn in 1974 [59]. KPN is a distributed computation model where a group of sequential processes communicate using FIFO channels. These channels are unbounded, i. e., they provide an infinite amount of storage and will never overflow. Processes are operating on local data and may read from and write to the FIFO channels. Writing to a channel is non-blocking due to the infinite amount of storage, reading is blocking. KPNs are often used for modeling embedded systems, since such systems often get a stream of input data that has to be processed. Such systems can be efficiently modeled using KPNs due to the streaming nature of the FIFO channels. Figure 2.3 shows a sample KPN with five processes that communicate using four channels.

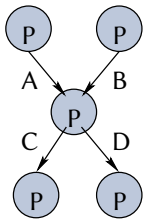


Figure 2.3: Example KPN with five communicating processes

Architecture Characterization

The models presented in the previous paragraph require a detailed knowledge of the underlying architecture. Examples for such characteristics are the number of clock cycles needed per local operation, the bandwidth and latency of memory access to different kinds of memory and the latency introduced and bandwidth achievable when accessing the network. Vendors usually provide such performance characteristics, but these values tend to be optimal peak values that are hardly achievable in real applications. To overcome this issue, several benchmarking tools exist for gathering such parameters. The CPU's raw performance can be measured by implementing a small application that performs a huge number of operations on a small amount of data. The data in such benchmarking applications should fit into the L1 cache of the CPU, so the measurements are not influenced by access to different memory locations.

Characterizing memory access and the communication network is more complex. On these media contention may occur due to several processors accessing the same memory or the network at the same time, increasing the latencies and reducing the bandwidths of these media. While these parameters may often scale linearly with the number of processors accessing the medium, especially the bandwidth will usually largely depend on the size of the data to be exchanged.

The STREAM [77, 76] and the RAMSpeed [22] benchmark are characterization tools that perform simple operations on a large continuous block of memory. Data is accessed incrementally as a stream of data, so the caching system is not able to accelerate the memory access. The benchmark is configurable regarding the size of the memory block to be accessed and can be used for measuring the uniprocessor bandwidth achieved as well as the bandwidth achieved in multiprocessor runs.

For characterizing the communication bandwidth between processors using message passing a wide range of benchmarking tools exist, for example Netperf [16] for measuring the TCP/IP communication performance of a beowulf cluster. The OpenFabrics [19] Infiniband stack commonly used by several vendors of Infiniband hardware provides several tools for measuring the low-level performance of an Infiniband interconnect, similar tools are available for other interconnects. A more general approach is the Intel MPI Benchmark (IMB) [73] that relies on MPI and measures several parameters such as bandwidth and latency of unidirectional or bidirectional communication and the time needed for distributing data, for synchronizing the processors, for performing a reduction operation and many more. The benchmarks are performed with many different data sizes that are transmitted, giving a detailed insight of the performance values that can be achieved on the actual system.

Additionally, several benchmarks exist for computational kernels often used in HPC, such as the Linpack [91] benchmark solving a dense linear equation system which is also used for the ranking in the Top500 list [25] of the fastest HPC systems in the world.

Other examples are benchmarks performing sparse matrix calculations [92], FFT [54] and many more. While such kernel benchmarks do not directly characterize parameters of the actual system, they may act as a basis for a first estimation of the performance achievable when implementing similar applications or even applications that make heavy use of such kernels. Several benchmarking pages exist for comparing systems' performance achieved in such benchmarks, for example the PC^2 benchmark site [99].

2.2.2 Analytical Performance Estimation for Reconfigurable HPC

While the methods presented above greatly aid the developer in tuning the runtime performance of applications executed on commodity large-scale HPC systems, the situation is more complex when regarding reconfigurable accelerators. The analytical models described before provide an abstract architecture model simulating the target system and executed the application model on this target machine under certain constraints. FPGAs do not provide a predefined set of execution units that can execute arbitrary code. The architecture has to be defined by the user based on the actual execution model. Since the design and implementation of reconfigurable accelerators is a time consuming task, models are needed to decide whether and which segments of the application are suitable for acceleration. The following paragraphs discuss some related approaches to characterize algorithms and identify such segments.

Characterizing Algorithms by Calculating the Computational Density Function

In [118], Steffen presents a scheme for characterizing algorithms and computing hardware regarding the suitability of the hardware for the algorithm. The method does not try to predict the performance of a specific design implementing the algorithm on a specific hardware accurately. Instead it is a pre-design analysis of code or algorithms to decide whether porting the software to an FPGA system is even worth the effort. For this purpose, a *computational density function* ρ of the algorithm is calculated.

The method assumes calculations to be performed on an abstract processor which has local memory, analogous to a cache in a real processor. First, an input data set of m operands each of size s has to be transferred to the local store, resulting in a complete transfer size of $M = m \cdot s$. Z computations have to be performed, each one requiring v operands. The local store size is defined to be of size α, the computational density function $\rho(\alpha)$ is the number of computations per byte that can be performed by the abstract processor. The model assumes that the local store is filled once and all possible computations on the data transferred are performed. After this processing, the next block of data is transferred and processed. This procedure is repeated until the complete problem M is processed.

In this model, the larger the local store size α is, the more computations per byte

can be performed before more data is required. If $\alpha > M$, then the complete problem can be transferred to the local store for computation in one step. The computational density function is formulated depending on the number of operations η that are possible with α bytes of data: $\rho(\alpha) = \frac{\eta(\alpha)}{\alpha}$. The η function has to be defined by the designer, requiring detailed knowledge of the algorithm. Several methods for gathering this value are presented in the paper [118].

Using this computational density function and hardware parameters like the memory size μ, the bandwidth β and the latency λ, the maximum throughput σ for the abstract processor can be calculated and compared to the measured value for the real processor. This comparison supports the developer in deciding whether the algorithm is applicable for FPGA acceleration without even requiring a rough idea of the final FPGA design.

RAT - the RC Amenability Test

The RC Amenability Test (RAT) presented by Holland et al. in [66] and [67] is a methodology for determining an application's suitability for FPGA acceleration. The method consists of an analysis of algorithms or existing legacy code combined with some computations. Three factors for the amenability of an application to hardware are considered: throughput, numerical precision and resource usage.

The goal of the throughput analysis is to estimate the performance of the application implemented on a reconfigurable computer based on parameters like the interconnect's speed, the amount of data to be transferred, the number of operations performed per data element and the clock frequency of the accelerator. A set of equations is given for calculating the time needed for communication and computation based on these parameters, as well as the overall runtime, utilization and speedup of the accelerator.

In addtition to the throughput analysis, RAT provides methods for estimating the numerical precision an accelerator has to provide. General-purpose processors have fixed-length data types and a fixed floating point format and will often use higher precision than actually needed for specific algorithms. In FPGA accelerators, the operators and their precision can be freely defined, taking parameters like performance, numerical errors and resource usage into account. RAT relies on third-party research and tools for evaluating the required precision and takes the results into account when calculating the throughput achieved and area used by the accelerator. Examples for such third-party tools are presented in [37, 41, 39, 58].

Resource utilization is the third property that is considered by RAT. However, for this step no generic formulae are published. The actual estimation is based on the user's expert knowledge of the target FPGA's architecture and the algorithm. The RAT methodology starts with a kernel identification and a design created on paper. A throughput analysis is performed followed by a resource test and a precision test. Using the results, the original design on paper is updated and the test starts from the beginning. This procedure is

repeated until no further optimizations are identified. The method results in an estimation of the achievable speedup which then can be used to decide whether implementing the accelerator is worth the effort or not. In [65] the authors present the integration of RAT into the RCML environment for estimation modeling of reconfigurable computing systems [101].

A Modeling Approach for Complete HPRC Applications

The two approaches presented in the previous two sections only consider the suitability of a single algorithm or compute kernel for FPGA acceleration. However, for estimating the performance achievable for complete applications running on multiple processors and multiple FPGAs, the estimates gathered from these techniques have to be integrated into more comprehensive models.

The modeling and analysis approach presented by Smith and Peterson in [112] and [113], as well as by Smith in [111] is such a comprehensive model. As a basis, it assumes a system consisting of multiple workstations connected to a cluster, each one equipped with one or several reconfigurable hardware components. The method assumes synchronous iterative algorithms, in which several tasks can be executed in parallel. The tasks synchronize after each iteration, just like in the BSP model [125]. Some of the tasks are executed on the worstations' CPUs, the others are executed in the reconfigurable hardware. Using these properties and parameters like communication bandwidth between the nodes, execution time of the tasks in hardware and software, communication time between CPU and FPGA, load imbalance between nodes and amount of serial computations, formulae are developed for calculating the complete runtime of the application on the target system.

The hardware tasks themselves are expected to have a deterministic runtime, e. g., not containing any decision loops. The runtime can in this case be calculated by counting the clock cycles used and dividing this value with the processor's clock frequency. More complex tasks would need additional effort and could be analyzed for example by the methods discussed in the previous two sections.

Tool Supported Performance Estimation

Corresponding to the profiling tools available for commodity workstations, recent research activities explored the utilization of automatic partitioning tools for heterogeneous computing. In [115], Spacey et al. introduce the 3SP Design Space Exploration System. 3SP characterizes software and automatically partitions it for execution on heterogeneous architectures. Hardware characterization parameters, such as cycle time, number of parallel execution units, bus latency, bandwidth etc. are provided in a user-initialized configuration file. The application is characterized using the 3S instrumentation and characterization framework presented in [116]. With this information, the 3SP tool seeks an optimal

partitioning between CPU and accelerator.

Another approach presented in [79] by Kenter et al. is based on the LLVM (Low Level Virtual Machine) compiler infrastructure. Here, the application to be analyzed is compiled into the LLVM intermediate code (IR) representation. The application is modeled as a set of instructions classified into load/stores and calculations which are grouped into a set of basic blocks. The architecture model consists of a CPU and an accelerator with private L1 caches, a shared L2 cache and memory, and a low latency control interface between CPU and accelerator. Basic blocks can be mapped to the CPU or to the accelerator. The architecture is characterized by parameters such as cache sizes, execution efficiencies of CPU and accelerator (e. g., number of clock cycles spent per instruction) and so on. The application is executed on a commodity workstation and the framework monitors the memory access scheme of each basic block. With these information, the overhead of moving basic blocks to the accelerator in terms of data transfer times is calculated — together with the estimated execution efficiency, the overall speedup is calculated afterwards.

2.2.3 Bottleneck Identification and Optimization of Reconfigurable Accelerators

When the initial performance analysis resulted in the perception that the target application is suitable for FPGA acceleration and an initial hardware/software partitioning was performed, the design and implementation process is started. While some bottlenecks can often be identified during the simulation phase, due to the reduced problem sizes usually used in this phase other bottlenecks may be only found in the real system running in the FPGA.

Software running on commodity CPUs can be analyzed by a wide variety of performance analysis tools as described above. For FPGA-based accelerators, profiling tools are currently not as widely accepted as for software. Vendors usually provide methods for reading the state of a user-defined set of signals during runtime using JTAG or similar methods. Since these methods do not count the number of events occurring on the signals monitored, but only trace the selected signals, they are not directly usable for profiling. Instead, counters have to be inserted manually which have to be monitored by the JTAG interface.

In [80], Koehler et al. propose a framework for profiling applications running on high performance reconfigurable computers. A software parses the source HDL code and enumerates signals, ports, variables and other data. The data to be monitored can be selected automatically — based on user settings — and modified manually. The software then generates a modified type of the original HDL code with instrumentation code inserted for the data to be monitored. In the software implementation, a seperate thread is started that polls the performance counters for gathering the data. In [47] and [48] the authors extend this approach by expanding the framework to support the challenges occurring in High Level Language Synthesis.

2.2.4 Reconfigurable Hardware Characterization

Only little related work exists regarding the architecture characterization of reconfigurable hardware. The achievable clock speed of simple operations like add or subtract usually greatly depends on the actual implementation, with parameters like the data width, cycles per operation, latency, etc. Since these implementation parameters depend on the application to be implemented, no standard benchmarks are available. Instead, designers evaluate implementations using different parameters as needed by the application.

The maximum communication bandwidth between CPU and FPGA as well as between FPGA and different kinds of memory is a parameter that is primary defined by the architecture and the access scheme, just like the memory and network bandwidth and latency in traditional HPC systems. Due to the lack of standard interfaces and communication channels, no standard benchmarking tools can be found. In [105] Schmidt and Sass present a characterization of the memory access performance on a widely used standalone FPGA board. Several cores are connected to the memory using a shared bus and access this memory using different patterns.

2.3 Design Implementation, Verification and Optimization

Reconfigurable accelerators provide a great flexibility. This flexibility also produces several challenges in the accelerator design. While designing such accelerators using traditional hardware description languages like VHDL or Verilog provides full control of the design, this is also the most complex way to go. Several approaches try to harness these challenges, including the work flow presented in this work. This work does not try to replace all of these approaches but may also integrate these other approaches.

An example for methods that simplify the designer's work are libraries of cores. Such libraries contain cores for many different functions that are usually customizable to the concrete applications needs. A graphical user interface is often provided for setting the core's parameters and generating a customized variant. Examples of such libraries and generators are the Xilinx Coregen, which is part of the Xilinx design environment, the Altera Megawizard providing an equivalent functionality for Altera devices, and the vendor neutral grlib [27]. The OpenCores Website [18] also provides a wide range of freely available cores with different functionalities.

2.3.1 High-Level Language (HLL) Synthesis

High-Level Language Synthesis is an approach for overcoming the time consuming and error-prone task of describing RTL hardware using common HDLs like VHDL [82] or

Verilog [96]. With the growing acceptance of reconfigurable hardware for accelerating high-performance computing, such approaches gained a new dimension of attention.

There are various projects that studied the generation of hardware out of standard high-level languages, usually based on C or C++ with some limitations and extensions. Table 2.1 gives an overview of well-known HLL compilation languages and tools. All these tools provide different features and levels of abstraction from the real hardware, and support different subsets or supersets of ANSI C/C++.

This results in that the languages are not compatible to each other: once decided to use a specific language designers are forced to stay with that language or completely reimplement the design. This disadvantage becomes even worse due to the fact, that not all of the languages and tools support generic hardware. For example SRC's Carte is only available on machines provided by SRC, Dime-C only for Nallatech's FPGA platforms. Other tools, like Impulse C and Handel-C provide board support packages for different FPGA platforms, including the generation of standard HDL code from disregarding the concrete platform.

Tool		Provider
Carte	[117]	SRC Computers
Catapult C	[84]	Mentor Graphics
DEFACTO	[50]	Univ. of South. California
Dime-C	[89]	Nallatech
Handel-C	[85]	Mentor Graphics (formerly: Celoxica)
HardwareC	[81, 49]	UC Stanford
Impulse C	[72]	Impulse Accelerated Technologies
Mitrion C	[87]	Mitrion
NapaC	[60]	National Semiconductor
PRISM	[28, 29]	Brown University
ROCCC	[62]	University of California at Riverside
SA-C	[75, 64]	Colorado State University
Sea Cucumber	[122]	Brigham Young University
SPARK	[98]	UC San Diego
Streams-C	[61]	Los Alamos National Labs
SystemC	[95]	Open SystemC Initiative (OSCI)

Table 2.1: Selection of well-known HLL synthesis tools

A complete discussion of all these languages is out of the scope of this thesis, only some basic properties of commonly used tools will be discussed. Detailed information on the languages can be found in [40], where the authors present different compilation techniques for reconfigurable hardware. For an in-depth comparison of some of the languages listed including benchmarks refer to [68].

SystemC is based on standard C++ with some extensions implemented as class libraries, providing support for HW/SW modeling. Behavioral and RTL designs can be modeled; structural hardware modeling is supported using modules, ports, interfaces and channels. Since SystemC is completely based on standard C++, standard compilers can be used for generating an executable out of the source code by linking the SystemC library. The executable can be executed directly on a workstation for simulation. In addition, several synthesizers are available converting a SystemC designed system into VHDL or a netlist. SystemC is an open standard developed by the Open System C Initiative (OSCI) and approved by the IEEE. These properties made SystemC becoming a very popular language for system level verification which is supported by many tools from different vendors.

In Mitrion C, the compiler does not directly generate any hardware. Instead, the C source code is mapped to a virtual processor, the Mitrion Virtual Processor (MVP). In contrast to regular soft core CPUs available for synthesis in FPGAs, this processor does not come with a fixed instruction set and architecture, but is automatically tailored to efficiently execute the specific application. Depending on resource constraints, computational units can be duplicated to exploit parallelism.

Another method is the one performed by Celoxica's Handel C. The C code implements an algorithm, but the code is directly translated into VHDL/Verilog/SystemC or a netlist. Parallelism can be defined by using pragmas, resembling the method openMP uses for implementing parallel applications for shared memory systems. The implementation is cycle-accurate, all operations occur in one clock cycle, but the compiler may analyze and optimize the code to improve timing.

Streams C and Impulse C, which is an advanced and commercialized variant of Streams C, are more resembling the MIMD (Multiple Instruction stream, Multiple Data stream [55]) programming compared to traditional software design. Applications are implemented using multiple sequential, independent processes which may run concurrently on the FPGA. Communication and synchronization is performed using streams. The output of the compiler is a parallel architecture, implemented as generic or FPGA specific VHDL code.

Regarding the underlying model of these compilation techniques, the Streams C and Impulse C approach resembles the model presented in Chapter 3 in many points. Thus it is an interesting alternative for implementing part of the cores of an accelerator in the case the IMORC architectural template introduced in Chapter 4 of this thesis is used.

2.3.2 Visual Design Entry

With visual design entry tools, designers can implement systems on an FPGA partially or completely in a graphical way. Analogous to HDL designs, structural and RTL designs can be generated this way. Designs are drawn as circuit diagrams with design units represented as boxes, which are connected using wires. Design units can be simple gates or flipflops,

as well as more complex subdesigns which themselves are again implemented using a schematic, an HDL or any other appropriate way. Often, such tools also provide methods for modeling the behavior of design units using finite state machines or activity diagrams. After the design phase, the tools generate HDL code for implementation or simulation.

The number of such visual design tools is quite large. Altera includes a simple schematic editor in their Quartus design suite. A more advanced tool is the HDL designer [86] provided by MentorGraphics, which provides a complete graphical design environment for generating the system and documentation.

In the recent years several similar tools were developed with a focus on the generation of DSP systems. The tools are based on Mathworks' Matlab/Simulink tool [121] and generate synthesizable HDL code out of Simulink models. An example for such a tool is the HDL Coder, an extension for Simulink directly provided by Mathworks. Using HDL Coder, systems can be modeled in Simulink, Stateflow and embedded Matlab code, VHDL or Verilog code can be produced for synthesis.

Other tools based on Matlab/Simulink are Xilinx System Generator for DSP [127], Altera DSP-Builder [32] and Synopsys Synphony Model Compiler [120]. The tools provided by Xilinx and Altera support their own devices and include libraries of functions based on their own core generator tools (Xilinx Coregen and Altera MegaWizard). Synphony does not target a specific vendor's devices, similar to the Mathworks HDL Coder, it generates vendor neutral HDL code.

Since Matlab/Simulink also provides direct interactions to simulators like ModelSim, generated systems can be directly integrated into a testbench and simulated along with custom HDL code. While the modeling framework and architecture introduced in this thesis do not directly build on one of these tools, they may be well integrated into the tool flow as required. The schematic editors are appropriate tools for integrating multiple cores into a complete system. Datapaths and control structures are typically good candidates for being implemented by the state machine generators or the DSP generator tools.

2.3.3 Multi-Core System Generation

While the visual design entry tools are adequate for structural design if connecting multiple design entities using custom interfaces, several other tools are available for generating systems on chip using interconnect standards. The Xilinx Platform Studio is such a tool for generating systems based on the IBM CoreConnect [70] infrastructure. This consists of a Processor Local Bus (PLB) and a lower speed On-chip Peripheral Bus (OPB) (the latter is regarded as deprecated in the latest releases). Multiple of each of these busses can be used in a system, communication between busses can be performed using bridges. The bus is a shared medium for all cores connected to it. This leads to the fact that all connected cores have to communicate to the bus at the same clock frequency and use the same data width, and that all the time only one core may send requests to the bus and

only one may utilize the data bus. Usually, one bus is used for fast communication from processor or computation cores to memory, while a second bus that is connected to the first using a bridge is available for cores with lower bandwidth demands. The bus in such a scenario forms a central bottleneck. If cores need to access data at different widths, the interface to the appropriate bus has to perform a bitwidth conversion. If data buffering is needed, this also has to be performed by the interface or by the core itself.

A different approach is used by the Avalon infrastructure used by Altera's SOPC Designer. This infrastructure uses a technique called Slave Side Arbitration. Here, one bus exists for each core that can act as a slave, i. e. respond to read/write requests from other cores. Each master core that is connected to this slave core is connected to the appropriate slave bus. The fact that only cores connected to the same slave share one bus reduces the bottleneck introduced by the shared medium. Clock domain crossing bridges are available for directly connecting masters and slaves residing in different clock domains.

ARM's [9] AMBA is another popular communication infrastructure often used in SOPC design. While it is traditionally a bus based interconnect like CoreConnect, other topologies are also possible for gaining better throughputs. Wishbone [97] is an open infrastructure frequently used for designs hosted on OpenCores [18] that uses similar concepts like AMBA.

In [106] and [107], Smith et al. present a different approach for implementing systems on chip using multiple communicating cores. They use directed links for sending data from one core to another. Data buffering is done using FIFOs, communication control and data sending/processing are independent from each other. A dedicated unit is used as controller in each core which can be programmed for the communication scheme to be implemented. As a result most of the work is to be spent for the datapath design, only little effort is needed for the controller.

To support the interoperability between tools, the IP-XACT [38] format was created by the SPIRIT consortium. IP-XACT is a XML format to describe and configure IP. System generator tools like the Synopsys System Designer tool that support the XACT format can directly import such IP, the designer may configure and use the IP in his design.

2.4 Chapter Summary

Many different methods exist for assisting developers in generating efficient accelerators for high performance reconfigurable computing. Modeling techniques are used for estimating the speedups possible for specific algorithms and for generating an efficient design, simulation techniques help finding errors and bottlenecks before synthesis and methods are available for measuring runtime performance values and bottlenecks during runtime. Also, different approaches are available for accelerating the task of the actual implementation. None of the methods are generally suitable for all kinds of accelerators.

Depending on the actual architecture of the target design different tools may be chosen for the actual implementation. The success of a design greatly depends on the methods and tools used during modeling, design and implementation phase.

The remaining chapters present an alternative integrated modeling and implementation workflow using a custom communication framework that overcomes some of the limitations of existing frameworks. The modeling approach introduced does not focus on mapping algorithms to a specific target architecture, but on defining an architecture to which the desired algorithms can be efficiently mapped. Furthermore the modeling approach does not focus on single operations to be performed, but on the amount of data to be transferred between memories and execution units, and is flexible in terms of the level of detail an application is to be specified by the model. The modeling approach is supported by an architectural template that helps in implementing and analyzing the application modeled. The workflow introduced in this thesis however can coexist with modeling approaches discussed in this chapter — using the modeling approach, parts of the overall model may still be specified by a PRAM, KPN or other techniques.

3 Programming, Execution and Performance Model

This chapter discusses the design flow and the different kinds of models used within the IMORC framework. First, a summary of the programming model and the overall design flow is introduced. The different steps of the design flow are then presented in detail, along with a discussion of the different models applicable in each step.

3.1 Introduction

For analyzing the performance of algorithms running on commodity workstations, several theoretical models exist, as outlined in the previous chapter. These models typically consist of an abstract architecture, such as a set of execution units connected to a shared memory in the PRAM, and methods to specify the algorithms. The runtime of an algorithm can then be estimated by calculating the number of operations performed.

When generating FPGA based accelerators for high performance computing, the situation is more complex. While parts of the target architecture are usually predefined, such as the number of CPUs and their connection, the memory architecture and the number of FPGAs available, parts of the architecture have to be defined by the designer. The number and types of cores to be implemented in the FPGAs depends on the concrete algorithms to be executed, the interconnection network between cores depends on the communication needs of the algorithm. Moreover, the actual implementation of the algorithm greatly depends on the type of the core that actually has to execute the algorithm.

Unlike traditional models like the PRAM, the modeling approach presented in this chapter pays attention to these properties by providing a flexible way to not only generate an algorithmic model targeted at a fixed architecture, but it also provides a method for defining the architecture and execution model concurrently. The application is modeled as a graph of communicating tasks, while the architecture model envisions a network

of communicating cores. The modeling approach can describe the architecture of the accelerator and application at different levels of detail. Typically, one will start with a coarse grained architecture model describing the target system and a coarse grained execution model containing different tasks that are mapped to different resources of the target system. The execution model is then refined stepwise to better demonstrate the communication behavior of individual kernels. At the same time, the architecture model is refined based on the updated execution model, e. g., cores are added to the reconfigurable hardware in the architecture model.

During refinement, it is possible to switch to other modeling techniques at a certain level of detail. For example, if some tasks in the execution model are well suited for being executed on a PRAM, the concrete implementation of such tasks may be specified using the PRAM model. In the architecture model, such a task can then be mapped to a core resembling the PRAM architecture, such as a microcontroller.

3.2 The IMORC Programming Model

The IMORC programming model consists of an architecture model and an execution model. The goal of the architecture model is to characterize the target platform. The execution model specifies the application to be implemented. Both models can be created at different levels of abstraction, depending on the current stage of the design phase. The models are used for a first estimation of an algorithm's suitability for FPGA acceleration as well as for the design phase of the accelerator.

3.2.1 The Architecture Model

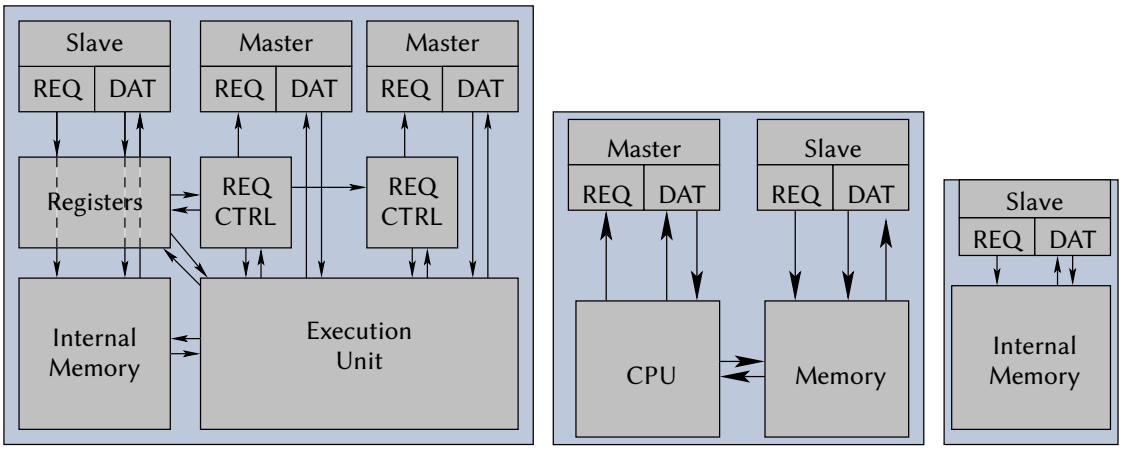
The architecture model underlying IMORC is a network of communicating cores. A core typically consists of some local storage, such as embedded memory blocks or registers, and an execution unit. Cores can communicate with other cores using an on-chip network. For accessing the network, each core provides an arbitrary number of communication ports, which are either master or slave ports. The network connects master to slave ports using links. Communication may be only initiated by master ports by sending a read or write request to the slave port. For write requests, the master has to send data corresponding to the request to the slave, for read request, the slave has to reply with a corresponding data packet. Master ports may connect to multiple slave ports, in which case addressing is required to select the target slave port of a communication request. Conversely, slave ports may connect to multiple master ports through an arbiter.

Figure 3.1(a) shows the block diagram of a typical execution (compute) core. The core comprises a slave and two master ports, each of which is separated into a request channel (REQ) providing control information and a data channel (DAT) for the actual data that has

to be transmitted. The slave communication interface is connected to a block of registers and to internal memory. A communication request controller exists for each of the master links, and each of them is connected to the register block for retrieving control information. An execution unit is responsible for performing the actual calculations and is connected to the data channels of the master ports. If required, the execution unit may also be connected to the register block and the internal memory block to store local data. Additionally, it may be connected to the communication request controller. This is, for example, useful if the core implements an iterative function where the number of iterations necessary cannot be calculated in advance, but depends on parameters that are calculated during runtime (e. g., an approximation algorithm). In such a case, the request controller has to be informed by the execution unit if further iterations are required or not.

The host processor (Figure 3.1(b)) is modeled as a simplified core that provides, depending on the actual target platform, one or multiple master ports and optionally one slave port. The core comprises a CPU that is able to execute arbitrary tasks and a local memory. The CPU can access this memory and send messages to the master ports. Additionally other cores may access the memory using the optional slave port.

Shared memory is modeled as a specific kind of core with no execution unit but a large amount of storage. A memory core provides exactly one slave port (see Figure 3.1(c)). When a master core sends a write request to a memory core, the corresponding data is stored in the core's storage, read request are replied with a response message containing the corresponding data from the core's storage.



(a) Compute core (b) Host processor (c) Memory core

Figure 3.1: Block diagram of an example compute core and a memory core

Figure 3.2 presents a sample architecture diagram including three compute cores, a host processor core and a memory core. The host interface core can directly access the slave port of core 0, which provides two master ports for accessing the slave ports of core 1 and

core 2. These master ports of core 1 and core 2 access the memory core, the master port of core 2 is additionally connected to the slave port of the host interface core.

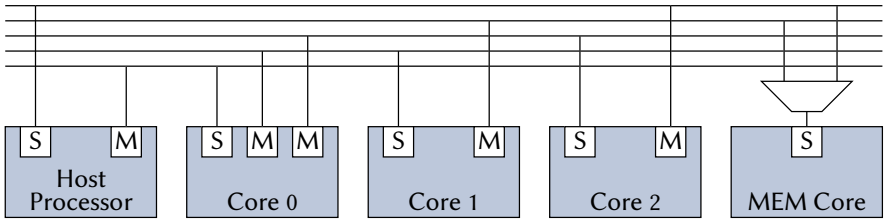


Figure 3.2: Sample architecture diagram with three compute cores, one memory core and one host processor core. S denotes a slave port, M denotes a master port.

3.2.2 The Execution Model

The intention of the execution model is to specify the application to be implemented and to support the definition of the architecture model. While the architecture model of IMORC comprises a set of cores that are connected to some kind of network, the execution model has to specify the actual behavior of these cores and their communication pattern. This model consists of a set of tasks, which are able to communicate to each other. Tasks are composed of a number of operations, which are classified into three distinct groups:

- Incoming communication operations, so the task has to process incoming messages,
- outgoing communication operations, so the task sends messages to other tasks and eventually has to wait for a response and
- local operations, such as the addition of two values.

Figure 3.3 shows an example for such a task. The tall box in the middle represents local operations performed by the task, the smaller boxes on the left and the right represent communication points with other tasks. Tasks can communicate with other tasks using messages. Messages can be read (`rd`) or write (`wr`), the first kind is used for requesting data from another task, the other one for sending data to another task. Read requests need to be followed by a response message (`rd resp`). Tasks can be modeled at different levels of abstraction, depending on the actual requirements. On a rather abstract level, the tasks computations may be described by pseudo code or a code segment in a high-level language. On a fairly detailed level, the computations may be expressed as a sequence of micro operations or RTL code. Tasks may directly operate on local memory. Shared memory accessed by multiple tasks is modeled as a separate task that communicates with the tasks accessing the memory.

Figure 3.4 shows a sample task graph with two computation tasks accessing a block of shared memory. Modeling memory as separate tasks simplifies the runtime analysis, especially in the case that multiple tasks access a shared block of memory and therefore

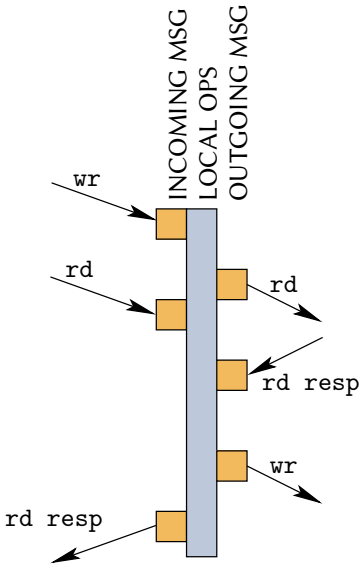


Figure 3.3: Diagram of a sample task

contention may occur. Additionally, due to the request-response nature of the communication model, tasks operating on streams of data can be modeled in a natural way. The communication subtasks send read requests to the appropriate memory task, and as soon as the data becomes available, the local operation subtasks start processing.

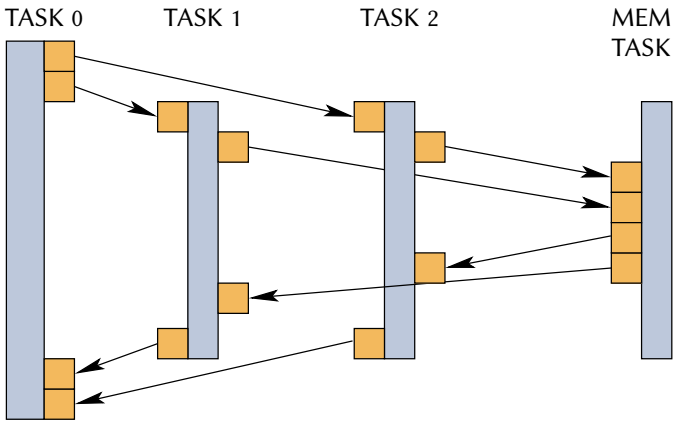


Figure 3.4: Sample task graph: task 0 starts two computation tasks, which in turn access a memory task

It has to be noted that the IMORC architecture and execution models are rather general and do not pose any restriction on the behavior of tasks or cores, respectively. For example, there is no need to enforce blocking reads on incoming messages as in the Kahn process

network (KPN) model, or to specify rates for the production of outgoing messages as in the synchronous data flow (SDF) model. In contrast to formal models of computation, IMORC provides more degrees of freedom but misses formally provable characteristics. However, formal models such as KPN and SDF can easily be embedded into IMORC by imposing corresponding rules.

3.3 Development Flow

Figure 3.5 shows a flow chart of the IMORC development flow. Accelerator development starts out with partitioning and initially mapping the application to the reconfigurable computing system. The communication time between the partitions and the time of computations is estimated, resulting in a first vague performance prediction. Repartitioning is performed until no further enhancements can be identified.

In the next step, the partitions mapped to the FPGA are refined for generating a detailed version of the task graph to be executed on the FPGA. Along with this refinement, the architecture model of the FPGA implementation is also refined for providing task-specific cores implemented in the FPGA. The iterative refinement process ends when the architecture model of the FPGA is detailed enough to complete the final implementation.

3.3.1 Partitioning and Initial Mapping

The goal of the partitioning and initial mapping phase is to perform two tasks: first, it gives a rough estimation whether the time consuming task of implementing an accelerator for the actual application is worth the effort. Second, the parts of the application that are to be mapped to the FPGA are identified.

The first step in this phase of the modeling flow is to generate a task graph representing the complete application. For a complete application, this task graph can be very complex. Accordingly the initial task graph should be very coarse grained and not present to many details. When generating a complete application from scratch, this may be a challenging task. However, often a software implementation of the original algorithm is already existing when considering reconfigurable computing for accelerating high-performance applications. The first decision in this case is, which parts of the application should be accelerated. For this, the application can be analyzed using one of the large number of available performance analysis tools. By profiling the most compute intense parts of the application can be identified. With this information an initial mapping of the application is generated (cmp. Figure 3.6). Each task is mapped to a CPU, an FPGA or to a memory location.

After partitioning and initial mapping a first performance analysis can be performed, for example, by estimating the amount of data to be transferred between the partitions.

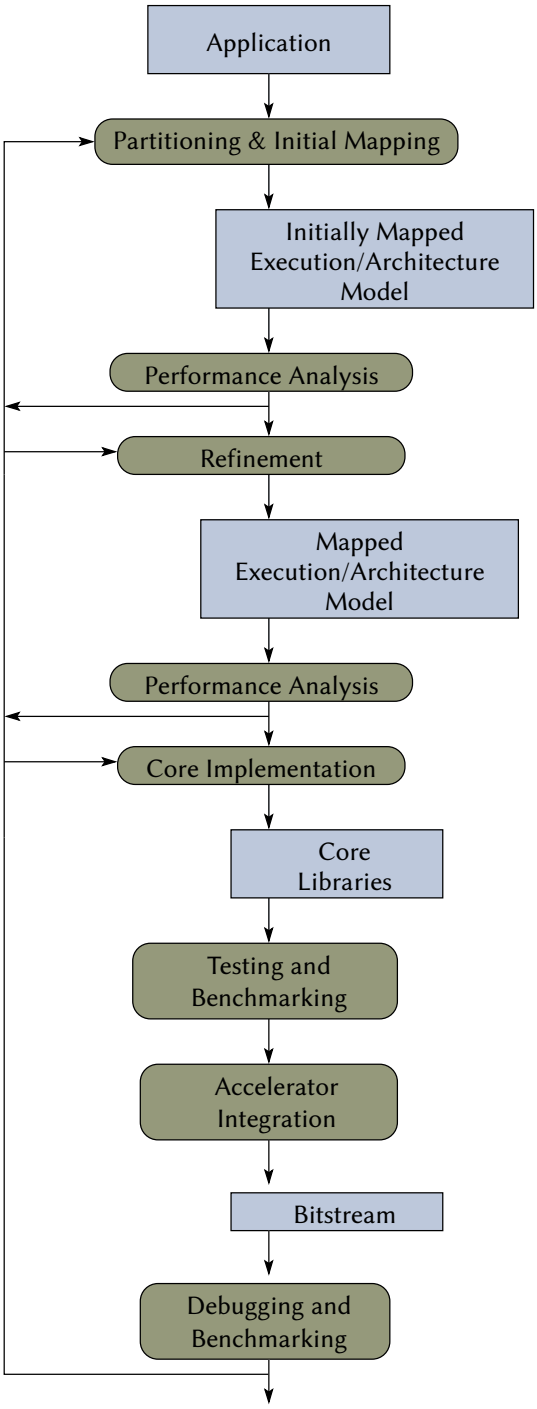


Figure 3.5: The IMORC modeling and implementation flow

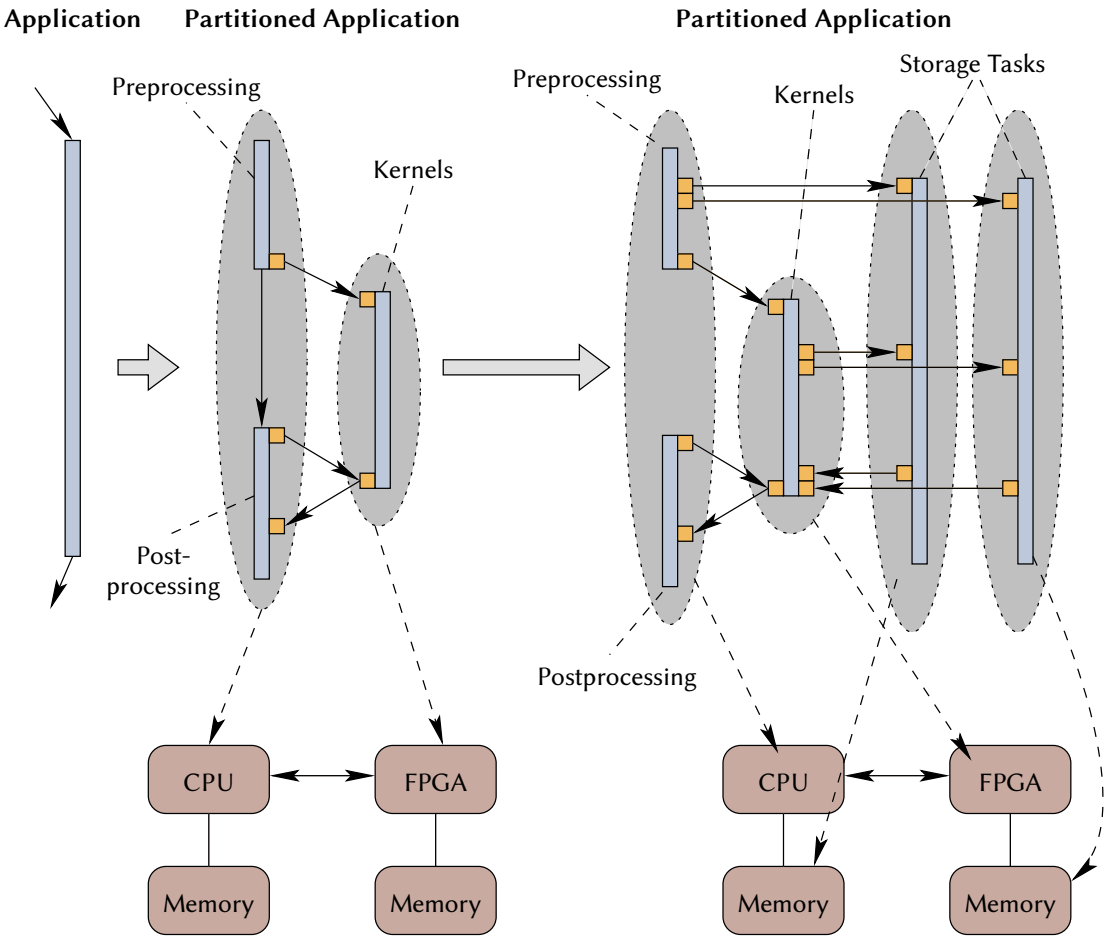


Figure 3.6: Initial partitioning of an application

Considering the mapped task graph in Figure 3.6, the initial runtime estimation can be performed by splitting the execution time into three phases:

1. in the first phase, the preprocessing task generates data and sends it to the two storage tasks,
2. in the second phase, the kernels mapped to the FPGA operate on this data and
3. in the third phase, the kernels synchronize with the postprocessing task.

The communication time can now be estimated by considering the amount of data transferred in each phase and the maximum communication bandwidth.

Such basic performance analysis can lead to new mappings, typically excluding certain kernels from the FPGA, adding further kernels to the FPGA (e. g., the data generation part

of the preprocessing task) or remapping storage tasks. The performance estimation can be improved by taking the communication scheme into account. Communication usually is faster when transferring large blocks of data than when transferring single values. With a concrete specification of the target architecture's communication performance for different transfer sizes and a more detailed specification of the application's communication scheme, more precise communication time estimates can be generated.

3.3.2 Task Graph Refinement

In the refinement phase, tasks are broken down to a more detailed level. While for tasks mapped to the CPU an efficient implementation often exists, tasks mapped to the FPGA have to be specified more in detail for generating the final implementation. They have to be split into multiple communicating tasks that actually form specifications for subsequent circuit design. An essential objective of the refinement step is to extract opportunities for exploiting parallelism.

An example for such a refinement is presented in Figure 3.7. The original task graph on the left consists of two tasks, a master and a worker. The master on the left sends the data to be processed to the worker task on the right. When finished with the first set of data the results are returned and the next set of data is transferred for processing. In the refined task graph, instead of waiting for the first set of data to be completely processed, data is streamed from the master task to the first worker task. This task performs some processing on the data and forwards the results to another worker task. The last worker task then returns the results to the master task. While the original task graph on the left splits the execution time into phases for communication and phases for computations, the refined task graph on the left performs both, communication and computations, in parallel. The utilization of the communication interface between the master and the worker tasks will be much higher than in the original task graph, resulting in an improved performance.

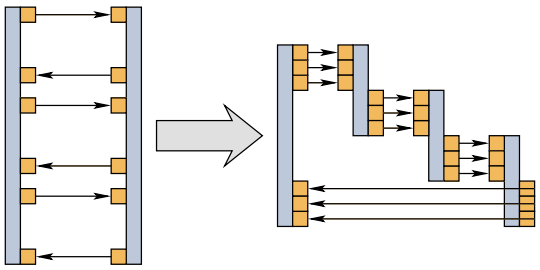


Figure 3.7: Example of a refinement by streaming data through a set of tasks

When streaming is not applicable but the original problem can be divided into subproblems that can be processed independently, the worker task can be split into several tasks all performing the same operations but on different data. This is especially useful if storage

tasks can deliver data faster than a single compute task can process it. For example, a dense matrix-vector multiplication task that processes the complete matrix row wise might not achieve a good throughput when implemented in an FPGA due to a low clock rate that is achievable by the arithmetic operators. However, in this example each row of the matrix can be processed independently of the other rows. An $N \times M$ matrix would result in M independent subtasks to be executed, each one performing N multiplications and accumulating the results. These subtasks are subject to the same performance penalties as the original task concerning the achievable clock rate, but since they are executed in parallel, the complete throughput of the resulting task graph is calculated by summing up the throughput of the individual subtasks.

A third possible refinement is the extraction of the request subtask and the datapath of a task. This method is helpful when the data to be processed depends on previous calculations. In many other models, as for example the PRAM, data is assumed to be directly accessible in constant time, so operations are directly applied to data. In the IMORC modeling approach, however, this dependency can be modeled as a subgraph. Requests for data and processing is split into separate tasks that are mapped to the same core — tasks requesting data are mapped to the communication controller of a core, tasks operating on the data are mapped to the execution engine of a core. Assuming for example a task implementing a sparse matrix-vector multiplication, where the matrix is often stored in Compressed Row Storage (CRS) format. In CRS the matrix is stored as three vectors, *val* containing all non-zero elements of the matrix, *col* containing the column id of each element in *val* and *row* specifying at which element of *val* a new row starts. The matrix stored in this way has to be multiplied by a vector *vec*.

Listing 3.1 shows the pseudocode of the sparse matrix-vector multiplication. The algorithm iterates over the rows in the matrix (line 13) and over all elements elements in each row (line 16) in the same manner as it would be done for a dense matrix-vector multiplication. The main difference is that the number of elements in each row is not fixed, but can be determined by vector *row*. The second difference is that the index into vector *vec*, which identifies the element of the vector that has to be multiplied with the current matrix element, is not simply incremented each iteration; instead, it is determined from vector *col*. While this method reduces the number of multiply/accumulate operations necessary, the random access of vector *vec* introduces a certain performance penalty if *vec* is large since random access to memory typically performs worse than sequential access.

Figure 3.8 presents the refined taskgraph of such a sparse matrix multiplication task which streams the data through the tasks. It consists of five storage tasks, one for each vector of the matrix (*val*, *row* and *col*), one for the vector to be multiplied with (*vec*) and one for the result vector (*res*). The request *col* task sends read requests to the *col* storage, the response stream (i. e., the stream of column indices corresponding to the matrix values) is forwarded to the request *vec* task, which sends corresponding read requests to the *vec* storage task. The resulting data stream contains as i^{th} element $vec[col[i]]$, which has to be

```
1  // Sparse matrix-vector multiplication
2  // Input:
3  //   n   - the number of rows
4  //   val - non-zero elements of the matrix, |val|=nv
5  //   col - column id of each element, |col|=nv
6  //   row - indices of val at which a new row starts, |row|=n
7  //   vec - vector to be multiplied with the matrix
8  // Output:
9  //   res - resulting vector
10
11 void spmv(int n, double *val, int *col, int *row, double *vec,
       double *res)
12 {
13   for(int i=0;i<n;i++)
14   {
15     res[i]=0.0f;
16     for(int j=row[i];j<row[i+1];j++)
17     {
18       res[i] = res[i]+val[j]*vec[col[j]];
19     }
20   }
21 }
```

Listing 3.1: C source code of the sparse matrix-vector multiplication kernel

multiplied by $val[i]$ (see line 19 of listing 3.1).

The values of the matrix are requested by the request *val* task. The *val* storage task and the *vec* storage task forward the requested values to the mult task. The mult task multiplies the matrix values with the vector values and forwards the results to the accumulate task. An additional request and storage task exists for the *row* vector of the matrix, the *row* storage tasks also forwards the corresponding values to the accumulate task. Based on the *row* values, the accumulate task calculates how many values have to be accumulated for each element of the result vector, performs this accumulation on the values received from the mult task and sends the results to the *res* storage task.

After these refinements, the communication scheme can be evaluated in much more detail as in the original mapping. As a result, with a detailed characterization of the available communication channels available on the desired target platform and the initial architecture mapping, the achievable performance may be calculated more precisely than before. The task graph now gives an overview of which tasks have to be implemented and how they communicate with other tasks. However, the concrete operations performed by the tasks to process input data and to generate output data are still not specified. The modeling approach presented does not dictate any rules of how to specify the concrete algorithm implemented by a compute task. This is due to the fact that different classes of

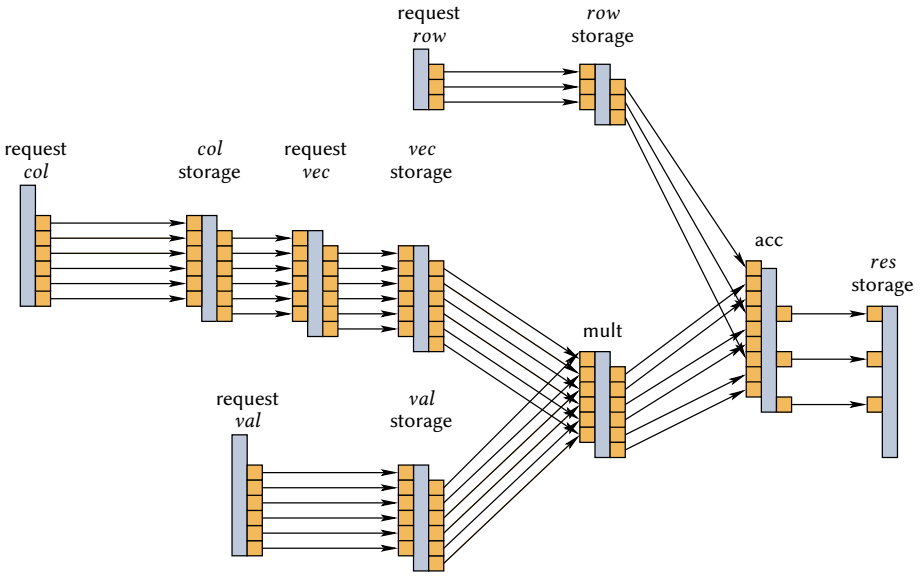


Figure 3.8: Detailed taskgraph of the sparse matrix multiplication kernel

algorithms may benefit from different modeling approaches. Tasks may be stateless, as for example the multiplication task in the sparse matrix kernel, or they are stateful, as the accumulation task that needs to observe the number of accumulations which have to be performed before writing the result to the output stream. Stateless tasks can typically be efficiently modeled by a data flow graph that can be mapped to a pipeline of operators during architecture mapping. The multiplication task in the example above would therefore be specified as a data flow graph with only one node, the multiplication operator. Stateful tasks like the accumulation task can be specified for example as a RAM with access to any kinds of network or as a finite state machine.

3.3.3 Architecture Generation

After the task graph has been refined to a reasonable level of detail, the initial architecture mapping needs updating. Each task has to be mapped to a certain resource, such as an execution core or a memory core. Multiple tasks may share the same resource, introducing a need for scheduling the resource. Naturally, the resource used for mapping a task has to be able to perform the operations specified by the task. Depending on the level of detail the task graph is represented at, a task may be mapped to a complete execution core or only to one or several resources of the core. Figure 3.9 shows an example for a mapping of the sparse matrix-vector multiplication task graph's compute tasks presented in Figure 3.8 to a single compute core. The request tasks of the task graph are mapped to the REQ CTRL modules of the generic core model as presented in Figure 3.1(a). The multiply and the

accumulate tasks are both mapped to the execution unit, which employs a multiplier that multiplies the values received from the two data channels containing *vec* and *val* and an accumulator that decodes the *row* values and accumulates the corresponding number of values as received from the multiplier.

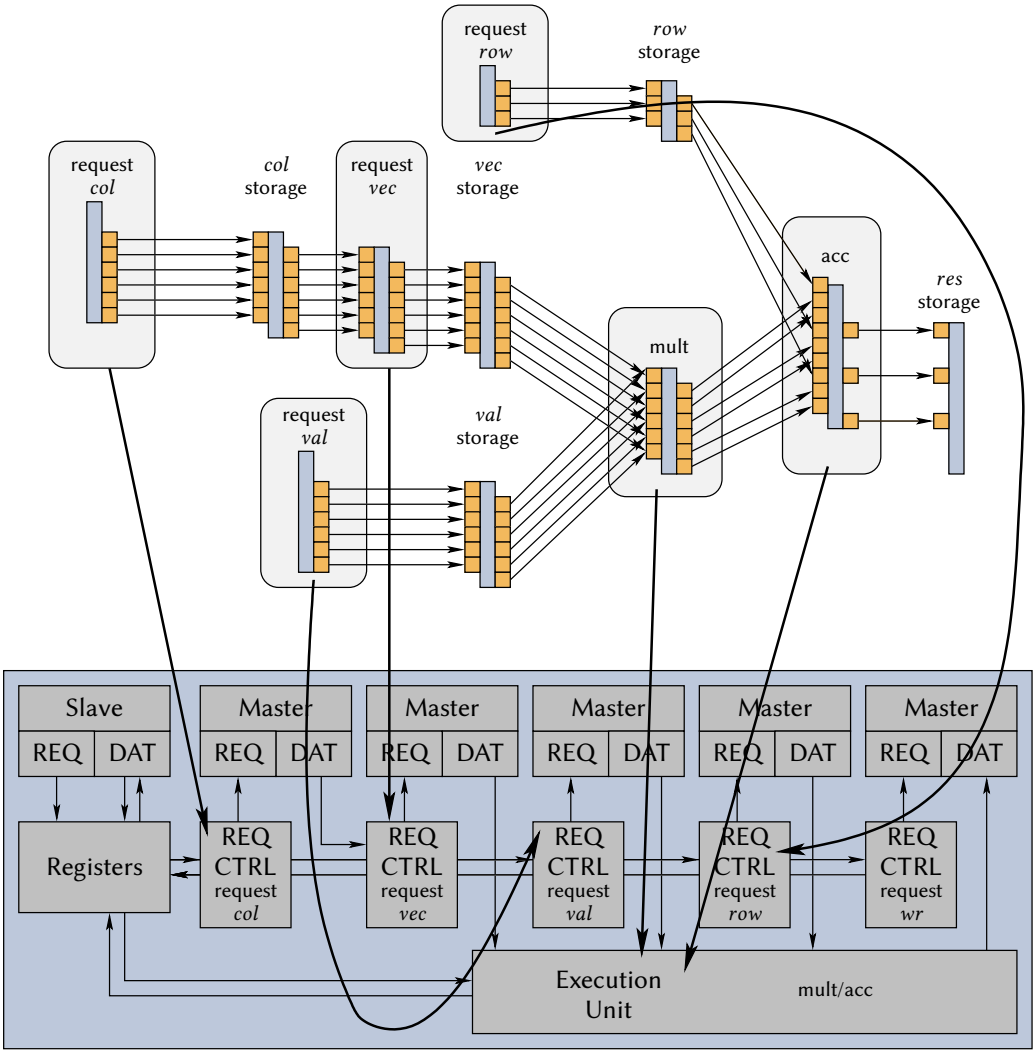


Figure 3.9: Mapping of the sparse matrix multiplication task graph to an architecture (one core with request controllers for accessing data and an execution unit implementing the multiply and accumulate tasks)

The storage tasks are mapped to the memory cores available in the target architecture or to additional memory cores that are implemented in the FPGA. Especially the connectivity of the available memory resources has to be considered, for example if host memory is

not directly accessible by the FPGA. In that case only storage tasks that are exclusively accessed by tasks mapped to the host CPU may be mapped to host memory. Another essential point that has to be considered is the amount of memory provided by each memory core and the performance of the resource. In particular if multiple storage tasks are mapped to the same memory resource, contention occurs that has to be considered.

To estimate the amount of contention and the bandwidth requirements of the computation tasks, a sequential runtime mapping has to be generated, where compute tasks mapped to the same resource are sequentially scheduled. This sequence is separated into execution phases so that the amount of data transferred by each core can be estimated for each computation phase. With an early estimation of the compute cores' runtime performance and their bandwidth requirements, the optimal mapping of storage tasks to memory cores may be found.

In the sparse matrix-vector multiplication example all compute tasks are mapped to different resources of the compute core and can be executed concurrently. Hence, such a separation can be omitted, i. e., the whole execution can be seen as one combination of communication and execution phase. With an $M \times N$ sized matrix with Z non-zero elements, Z values have to be read from *col*, *val* and *vec* storage, N values have to be read from *row* storage and N values have to be stored to *res*.

Another key point for estimating the execution time of such an algorithm is the scheme in that data is to be accessed. The bandwidth provided by the memory locations usually not only depends on the amount of data to be accessed, but also on the access scheme performed. Usually, data accessed in long bursts aligned to certain memory boundaries can be performed nearly at the peak rate the memory provides. However single-data transfers perform much slower. In the example presented above *col*, *row* and *val* can be accessed using such burst schemes, but accesses to *vec* are usually single-word transfers to locations depending on the value read from *col*. For gathering concrete performance values the memories have to be characterized not only regarding their peak performance values but also regarding different access schemes. The example presented above will only perform well if one of the following two conditions are met:

- *vec* is stored in a memory location with a very short latency (e. g. SRAM or even on-chip block memory) or
- the non-zero elements of the matrix are usually located in successive columns, enabling the multiplication algorithm to perform burst reads to *vec*.

If none of the previous two conditions are fulfilled, the implementation will not perform better than an implementation on a commodity CPU.

Assume all floating point values (i. e., the entries in *val* and *vec* and the result vector *res*) to be w_f byte wide and all integer values (i. e., the indices stored in *col* and *row*) to be w_i byte wide. The achievable bandwidth on memory k for a transfer size of bs byte per transfer is denoted as $bw_k(bs)$, $bw_k(\text{MAX})$ denotes the maximum bandwidth achievable with an optimal transfer size. If all vectors are stored to the same memory, the available

bandwidth is shared among the different storage tasks. All transfers are regarded as being performed sequentially, the runtime is estimated by summing up all times needed for data access:

$$t = \underbrace{\frac{Z \cdot w_f}{b_k(\mathrm{MAX})}}_{\text{access } val} + \underbrace{\frac{Z \cdot w_i}{b_k(\mathrm{MAX})}}_{\text{access } col} + \underbrace{\frac{N \cdot w_i}{b_k(\mathrm{MAX})}}_{\text{access } row} + \underbrace{\frac{Z \cdot w_f}{b_k(bs_{vec})}}_{\text{access } vec} + \underbrace{\frac{N \cdot w_f}{b_k(\mathrm{MAX})}}_{\text{write } res} \tag{3.1}$$

If the mapping plans the storage tasks to two different memory resources, memory k for the three vectors of the matrix and memory j for *vec* and *res*, the accumulated memory access times for both memories are summed up. The overall runtime estimation of the complete kernel is evaluated as the maximum of these two memory access times:

$$t = \max\left(\underbrace{\frac{Z \cdot w_f}{b_k(\mathrm{MAX})} + \frac{Z \cdot w_i}{b_k(\mathrm{MAX})} + \frac{N \cdot w_i}{b_k(\mathrm{MAX})}}_{=t_k}, \underbrace{\frac{Z \cdot w_f}{b_j(bs_{vec})} + \frac{N \cdot w_f}{b_j(\mathrm{MAX})}}_{=t_j}\right) \tag{3.2}$$

In the formula, t_k and t_j denote the time needed for all data accesses to memories k and j, respectively. If $t_k < t_j$, the overall execution time depends completely on t_j. One might argue that accessing *vec* happens after *col* was accessed, so both transfers influence each other. But, if $t_k < t_j$ then memory k will always be able to supply new input values to the request *val* task at the rate required to saturate the bandwidth of memory j. The latencies of the memories are disregarded in these calculations, since they only exert an influence to the overall runtime for very small problem sizes.

Mapping the algorithm to the XtremeData XD1000 platform, which is introduced in Chapter 4 and characterized in Chapter 5, data can be mapped to host memory or to external DDR SDRAM. Figure 3.10 shows the estimated performance in MFlops/s of different memory mappings to this machine. The estimations consider single precision floating point values (32 bit, denoted SP) as well as double precision floating point values (64bit, denoted DP). The graph shows three different mappings, first all data is stored in the DDR SDRAM, second all data is stored in host memory and third, the matrix is stored in host memory whereas *vec* and *res* are stored in DDR SDRAM. The graph assumes a banded matrix with a bandwidth of α, i. e., *col* consists of blocks of α consecutive values. This means that the vector memory can be accessed with a burst size of $bs_{vec} = \alpha \cdot w_f$ byte. However, on the DDR SDRAM as well as on the HyperTransport there are additional constraints for the addressing the data to be accessed. This results in that for a bandwidth of α, values from *vec* might only be accessible at a burst size of $\alpha/2$ on the actual memory.

Comparing the estimation with the benchmarks for the processor available in the XD1000 (SP, CPU and DP, CPU in Figure 3.10) might only perform better on the FPGA if all data is stored in the DDR SDRAM. Since the CPU can access host memory much faster

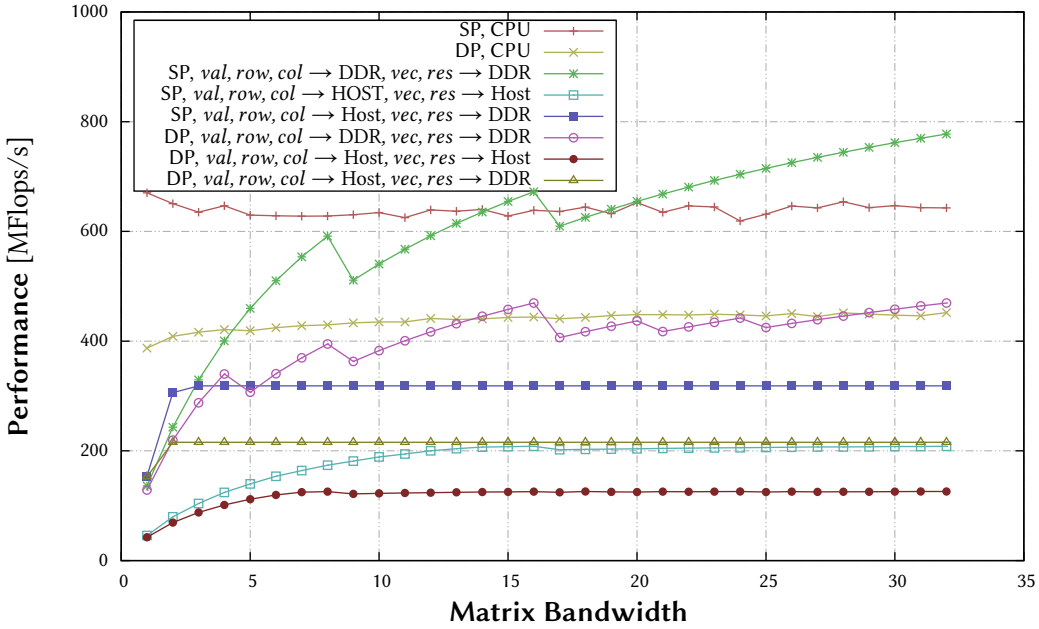


Figure 3.10: Performance estimation of the sparse matrix-vector multiplication kernel mapped to the XD1000. *val, row, col* and *res* are accessed with an optimal transfer size, the *x*-axis represents the number of elements out of *vec* transferred per request

than the DDR SDRAM of the FPGA, this will additionally reduce the overall application's performance. Hence, the proposed design will typically only be feasible when the tasks generating data are also mapped to the FPGA and provide a reasonable performance.

3.4 Chapter Summary

Modeling techniques are necessary for estimating the suitability of FPGA acceleration for high performance reconfigurable computing and greatly aid the designer in generating a concrete architecture for the accelerator. The model presented in this chapter does not aim at providing a detailed and accurate performance estimation but shall give a rough impression of the suitability of FPGA acceleration for specific algorithms. Furthermore, it helps the designer in generating an efficient design and to find potential bottlenecks in existing designs. Performance values of available resources such as memories and communication interfaces may be gathered from the vendor's specification or by performing microbenchmarks. After implementation, the model may be annotated with real performance values gathered out of the running system for further optimizations.

Contrary to most other modeling approaches, the modeling approach presented in this work does not highlight the execution time of operations to be performed on the data to be processed, but on the time needed for data access. The PRAM, for example, implies a shared memory that can be accessed by several execution units in parallel and in constant time. However, access times of real memories greatly depend on the actual type of memory. In shared memory systems, different levels in the memory hierarchy (e. g., L1-cache, L2-cache or main memory) provide different performance values. In distributed shared memory systems one has to distinguish between local memory and remote memory, additionally. Reconfigurable systems typically provide an even larger number of different storage options that have to be considered, such as host memory, off-chip SRAM/SDRAM or configurable on-chip memory. This modeling approach supports different well-known approaches for specifying the datapath of tasks, allowing the designer to select the specification method that seems most reasonable for a given task.

The model is flexible as it allows to specify the accelerator at different levels of detail as required in the current design phase. Cores that can be implemented straight-forward can be modeled at a very low level of detail, while others may be specified more precisely. This way the developer may choose the complexity of the model to gather the insights into the application he requires.

4 The IMORC Architectural Template

IMORC stands for Infrastructure for Performance Monitoring and Optimization of Reconfigurable Computers [4] and is an architecture template for implementing accelerators for high-performance reconfigurable computing. The main focus in the design of the IMORC architectural template was to provide an easy and straight-forward method for implementing accelerators with an architecture model as presented in the previous chapter. Therefore, it assumes an application that is decomposed into multiple communicating cores, which encapsulate computations and access to memory as well as to external communication interfaces. A key element of IMORC is its on-chip network for connecting these cores within the FPGA. To achieve high-throughput communication with minimal congestion, IMORC relies on a multi-bus architecture with slave-side arbitration.

This chapter discusses the elements of the IMORC architectural template including the communication infrastructure implementation and gives an overview of the provided utility cores.

4.1 Cores, Links, and Channels

IMORC cores access the on-chip communication infrastructure via ports. Corresponding to the architecture model there exist two types of ports, denoted as master and slave ports. Cores may provide an arbitrary number of master and/or slave ports, so they can directly be connected to multiple other cores. A link between two cores is formed by connecting a master with a slave port. Each link splits into three channels, a request channel (REQ), a master-to-slave channel (M2S), and a slave-to-master channel (S2M). The REQ channel is used for transmitting write or read requests from the master to the slave port. Data is transferred from the master to the slave port via an M2S channel, and from slave to master port via an S2M channel, respectively. A link must comprise at least the REQ channel

and can, additionally, specify M2S and S2M channels. Many parameters of the IMORC links are configurable at design time, Table 4.1 summarizes the available configuration parameters.

Parameter	Description
MASTER_ADDR_WIDTH	address width at the master port
SLAVE_ADDR_WIDTH	address width at the slave port
ADDR_OFFS	static offset added to the address
SIZE_WIDTH	width of the size field
USE_M2S	if true, write transfers are supported
USE_S2M	if true, read transfers are supported
SYNC	if true, synchronous FIFOs will be used
MASTER_WIDTH	bitwidth of the master port
SLAVE_WIDTH	bitwidth of the slave port
ALIGNED	transfers are guaranteed to be aligned to full slave words
REQ_DEPTH	depth of the REQ FIFO
M2S_DEPTH	depth of the M2S FIFO
S2M_DEPTH	depth of the S2M FIFO

Table 4.1: Configuration parameters of an IMORC link

Figure 4.1 details the signals for an IMORC link, basically consisting of a data bus and two handshake signals for each channel. The REQ channel uses a data bus `req` to transmit request packets and the two handshake signals `req_wait` and `req_wr` on the master side or `req_rd` on the slave side, respectively. The data to be written or read is then transferred over the M2S and S2M channels.

Table 4.2 presents the fields available on the REQ channel. The `CMD` field is a one-bit wide field specifying whether the current request is a data read or write. `ADDR` is used for setting the target base address of the current request and is interpreted by the slave port. For example, a memory controller core will use the destination address to access the attached memory, and a compute core could use it to select between a set of internal registers. Not all kinds of communication need to specify a target address, in which case this field can be omitted. This especially is the case if data has to be streamed from one core to another one, for example if cores in an image processing system are connected. In this case, the link can be configured to not support this field for saving area on the FPGA. The link can additionally add a static offset to the address field. This feature implements a rudimentary static virtual address space for each core and is useful if multiple memory tasks of the execution model are mapped to the same memory core in the architecture model. It allows cores to transparently access data mapped to arbitrary physical regions in the memory. The `SIZE` field is a required field, needed for determining the amount of data

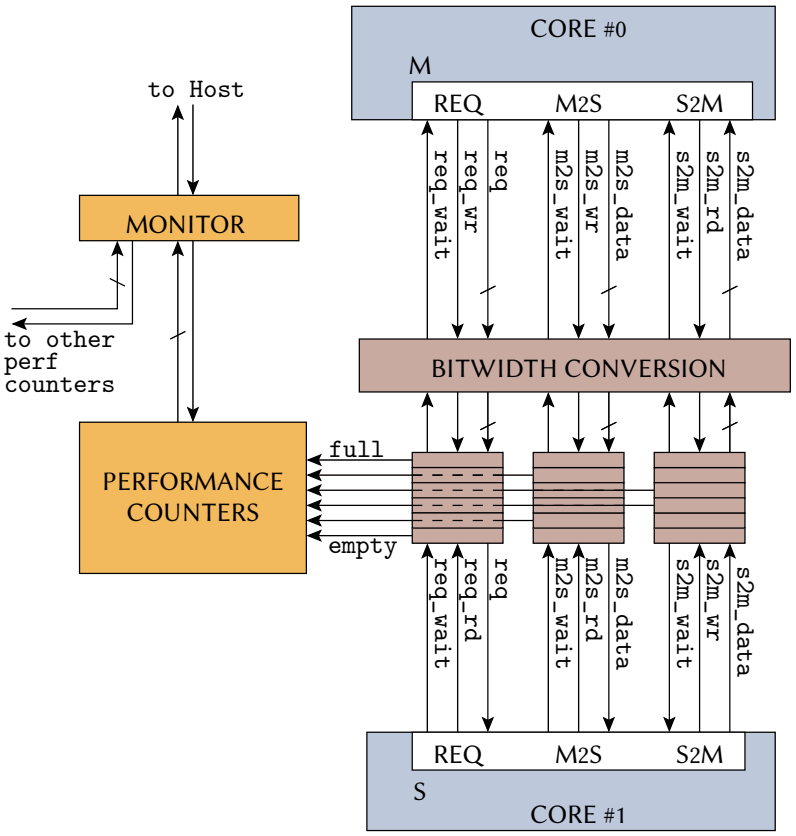


Figure 4.1: Block diagram of an IMORC link with its channels and signals

that is transferred. By default, the `SIZE` field specifies the request size in number of 32 bit words, but can be configured for other granularities if required.

IMORC inserts asynchronous FIFOs into each channel which allows each core to operate at its maximal speed in its own clock domain. This allows to operate each core at its top speed by choosing individual clock rates. It also enables slave cores like memory to serve several compute cores if the slave core's maximum bandwidth is greater than the individual

CMD	command field (RD or WR)
ADDR	destination address
SIZE	amount of data to be transferred (usually counted in 32 bit words, but configurable)
OPT	optional information decoding is up to the designer

Table 4.2: Request packet format

master cores' bandwidths. Optionally, the links can be configured to insert synchronous FIFOs if the cores connected share the same clock domain, which reduces latencies and area consumption of the links. Moreover, the additional FIFO storage in the network decouples the core's request task from the datapath. On the one hand, this simplifies the implementation of these two parts of the core, since requests usually may be posted independent of the datapath. On the other hand, this can often improve performance by hiding latencies of memories or other slave cores.

In case of different data bitwidths of master and slave ports, IMORC inserts a bitwidth conversion module into the link. The bitwidth conversion modules are placed on the master side before the FIFOs, which always have the same bitwidth as the slave ports. On the master side, data is expected to be aligned to the master port's width. That is, if the master port is n byte wide, every communication has to start at an address that is an integer multiple of n and the size also has to represent an integer multiple of n byte. The conversion from a wide master to a small slave word therefore is straight-forward. Every write to the master's M2S channel is translated into multiple writes to the corresponding FIFO, every read from the S2M channel is translated into multiple reads from the corresponding FIFO. Converting from small master words to wide slave words may be more complex under certain circumstances. If it is guaranteed that the master only performs transfers that are aligned to boundaries of the slave words data width, multiple writes to the M2S channel are combined to a single write to the corresponding FIFO, corresponding operations are performed for the S2M channel. If such an alignment cannot be guaranteed, the datapath has to decode the packet's request address and calculate an offset into the word of the slave's native size. The module can then generate sub-word write enable signals and transfer them to the slave. Alternatively, this job can be directly performed by the slave core.

As a consequence of these bitwidth conversion modules, a compute core with a 32 bit interface can be connected to a memory core with a 64 bit interface, and also to a 256 bit interface without any change in the compute core itself. This greatly facilitates reuse of cores and porting applications to different accelerator platforms. The integration of the FIFOs and the bitwidth conversion module into an IMORC link is shown in Figure 4.1.

4.2 Network Topology and Arbitration

An application typically comprises several IMORC cores connected in a certain topology. Each core can be equipped with an arbitrary number of master and slave ports. Hence, connecting master and slave ports in a 1:1 fashion is basically sufficient to build arbitrary topologies. However, as this approach requires cores with a rather high number of master and slave ports, IMORC supports 1:n, m:1, and m:n connections as well.

A 1:n connection allows a master port to address several slave ports at once. To this

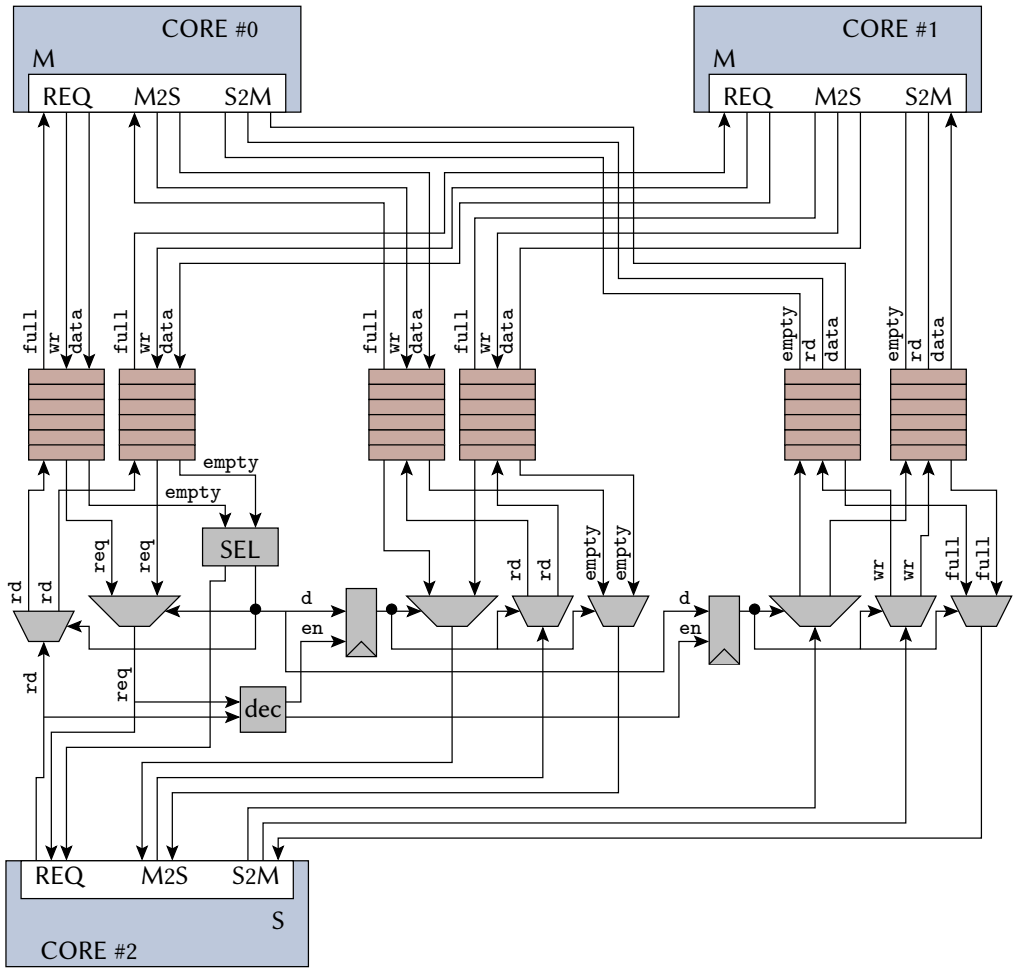


Figure 4.2: Arbitration module for a 2:1 connection

end, the signals `req_wr` and `m2s_wr` are turned into vectors. By selecting a subset of these write enable signals, the master may issue multicast or even broadcast request messages and data writes. The wait signals from the individual FIFOs also form a vector and are routed back to the master port. IMORC even supports the S2M channels in a 1:n connection with the restriction that a master can address only one FIFO to read from at any time. Read requests are also supported in such connections by turning the signals `s2m_rd` and `s2m_wait` into vectors. Since the bus `s2m_data` is shared among multiple links, the core has to make sure not to read data from multiple links at the same time. The links can be configured to tristate the `s2m_data` signal when `s2m_rd` is deasserted.

An m:1 connection allows a slave port to be driven from several master ports. IMORC employs slave-side arbitration and inserts an arbiter module right before the slave port. Figure 4.2 displays the IMORC interconnect for a 2:1 connection. In this figure, the optional

bitwidth conversion modules are omitted for the sake of readability. Additionally, for the same purpose the links' FIFOs are grouped into the three channels REQ, M2S, and S2M. A selector module SEL attached to the REQ channels of the links decides which request is served next. By default the selection is done in round-robin manner but it can easily be changed as needed, for example to introduce priorities. The selector module then informs the corresponding M2S or S2M channel's datapath about which request is processed next, along with the request's size. Data reads and writes performed to these channels by the slave core are forwarded to the appropriate link corresponding to the request packet. If required, *m*:1 and 1:*n* patterns can be combined to form an *m*:*n* connection.

4.3 Performance Counters

Optimizing the performance of an application consisting of multiple cores is not a trivial task. It requires to balance computation speed with communication bandwidth and to minimize contention for shared resources, e. g. for external memory. While the IMORC architectural template offers the designer freedom to address performance problems, e. g. by increasing data widths, replicating compute cores or replacing a compute core with a higher throughput version, selecting an appropriate remedy requires information about the dynamic behavior of the application.

To this end, IMORC provides performance counters attached to the FIFOs within the links and arbiter modules. For each FIFO, the number of full and empty events is counted as shown in Figure 4.1. A monitoring core reads and resets the performance counters in a user-defined time interval.

Figure 4.3 displays the implementation of the performance counters. The signal that is to be counted is connected to the input `event_in` of the performance counter and increments the event counter. Since the cores in a system may operate in different clock domains, the number of events is not sufficient to draw conclusions about the overall system. Hence, each performance counter additionally employs a cycle counter that is incremented every clock cycle.

When the monitoring core asserts the `sync` signal for one cycle, a register is toggled in the monitor's clock domain. The value of this toggle register is synchronized into the performance counter's clock domain and asserts signal `sync_int` for one clock cycle. When `sync_int` is asserted, the counters' current values are registered and the counters are reset. In register R3, `sync_int` is delayed by one clock cycle, the resulting signal sets register R4. R4 is synchronized back into the monitor's clock domain and there acts as a `valid` signal.

The registered counters are forwarded to the monitor, too. Note that no additional synchronization registers are used for these values. But since their values are set at the same clock cycle as register R3 is set and signal `valid` is asserted one cycle in the

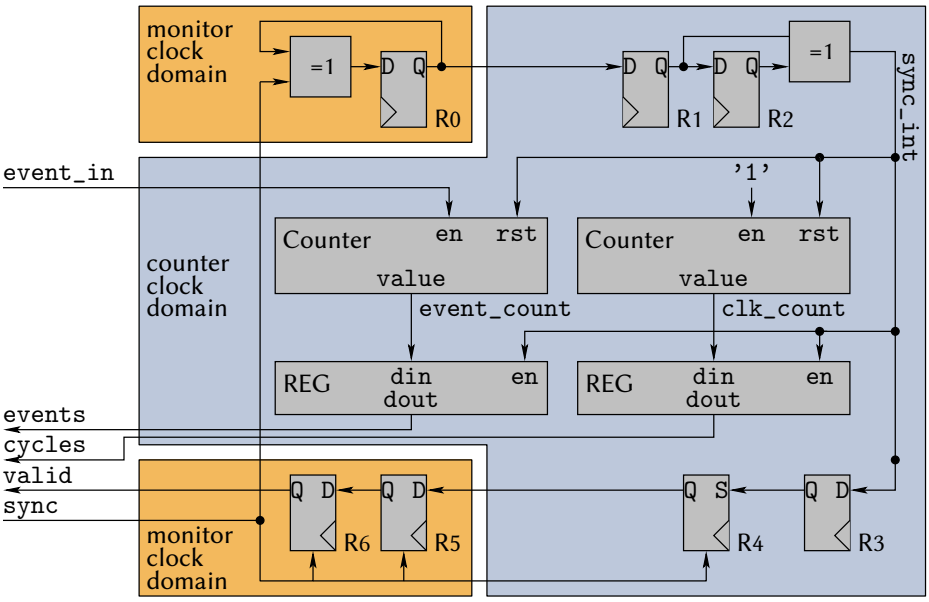


Figure 4.3: Diagram of a load sensor

performance counter's clock domain and two cycles in the monitor's clock domain later, the output values can be regarded as stable when `valid` is asserted.

All load sensors in the system that are observed have to be connected to a system specific monitoring core, as depicted in Figure 4.1. In user-defined time intervals the monitoring core asserts the `sync` signal and waits for `valid` to become asserted. Then, it reads the number of events and cycles and forwards these values to the monitoring PC.

Using the performance counter infrastructure, designers can monitor the dynamic behavior of an application and gather information about the cores, e.g. when they start or stop processing or how much bandwidth they use on different channels. Additionally to the performance counters available in IMORC's links the designer may integrate custom performance counters in his cores to monitor other signals.

4.4 Utility Cores

Besides the on-chip interconnect IMORC also provides several infrastructure cores, such as memory controllers and a host interface core, which are frequently needed in reconfigurable accelerators. Additionally, several supporting cores often needed are provided for simplifying the generation of cores, such as IMORC-to-Register interface cores, request generator cores and farming cores. While a core in IMORC usually only communicates using the REQ/M2S/S2M channels, some of the utility cores presented in this section are intended to be integrated in such cores. Hence, they typically only implement a subset

of the three channels and additionally provide sideband signals to communicate with the core that integrates the utility core. The following paragraphs give an overview over these cores and their features.

4.4.1 Host Interface Cores

A host interface core is needed by every accelerator to be able to communicate with the host. Accelerators can be attached using a wide variety of different host interfaces, such as PCI, PCIe, HyperTransport and many more. The host interface core is responsible for translating the protocol of the host interface to IMORC. The host interface core serves up to four different purposes, depending on what the actual protocol of the host interface supports. First, it is used for sending job information from the CPU to the FPGA. This usually incorporates one or several transfers of a small amount of data from the CPU to the FPGA accelerator. The second purpose is to transfer large amounts of data from the CPU to the FPGA, which is usually performed in large bursts for achieving high throughput. Third, the host interface may or may not support direct FPGA access to the host memory, which can also be supported by the host interface core. Fourth, the FPGA may be able to send interrupts to the host CPU. Figure 4.4 illustrates the interface of a host interface core with one IMORC master port and one IMORC slave port. Communication events initiated by the host are transformed into IMORC packets and made available on the master interface. Other cores may access host memory or send interrupts using the slave interface.

4.4.2 Memory Cores

IMORC provides different kinds of memory cores. First, an interface to on-chip memory is provided, written in synthesizable VHDL. A drawback at this point is, that current vendors' implementation tools are only able to infer simple types of on-chip memory from VHDL — these are single port or dual port memory without byte-enable signals. Since for high speed access wide memories are preferred in many cases, making subword writes necessary, IMORC also provides cores that explicitly instantiate the appropriate memories. However, this method is only portable when targeting the same FPGA vendor's devices.

IMORC also provides access to off-chip memory. The appropriate interfaces instantiate the FPGA vendor's memory controller cores and add some logic for translating the interface to IMORC. The third kind is host memory, as stated in the previous paragraph. When the host interface core supports direct access to host memory, it provides a separate IMORC slave port that can be accessed by the IMORC infrastructure on the FPGA. Current systems usually use virtual addressing for the host memory, so address translation may be needed in the interface core to generate a physical address.

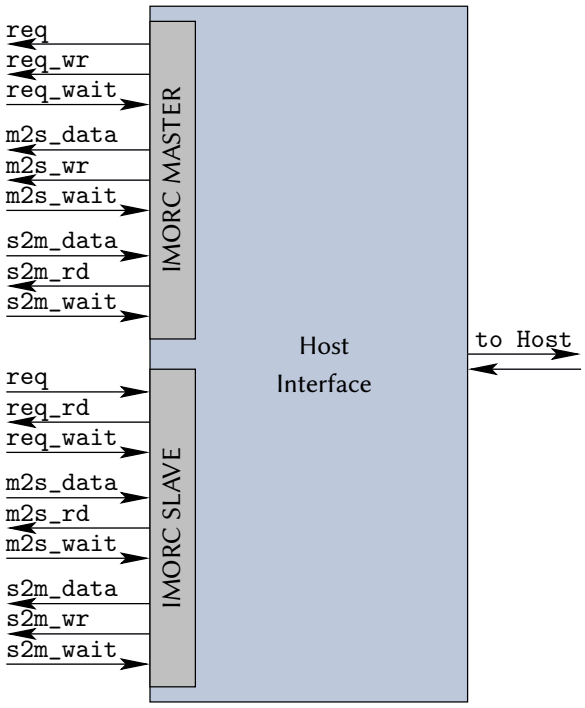


Figure 4.4: Sample host interface core with one IMORC master and one IMORC slave port

Access to these kinds of memories is completely transparent to the accelerator cores. Since all kinds of memories are accessed using a standard IMORC interface, from a core's perspective there is no difference between on-chip, off-chip and host memory.

4.4.3 Request Generator Cores

The request generator cores provide a method for generating different kinds of IMORC request sequences which are often needed by different kinds of cores. Instead of implementing specialized request tasks manually each time, the request cores can be instantiated and configured for generating the requests needed. A simple form of the request generator is the streaming request generator (cmp. Figure 4.5). It is configured with a command (read or write), a base address, the amount of data that has to be transferred and the size of each request. Then it starts posting the appropriate sequence of requests. Sending requests can be interrupted, for example if different requests have to be inserted by other request generators.

A more sophisticated request generator can inject a sequence of read and write requests to the same channel. Different base addresses, amounts of data and request sizes can be set for read and write requests. The sequence of reads and writes can be programmed by two further parameters, encoded as bitvectors. The first one represents the initial sequence,

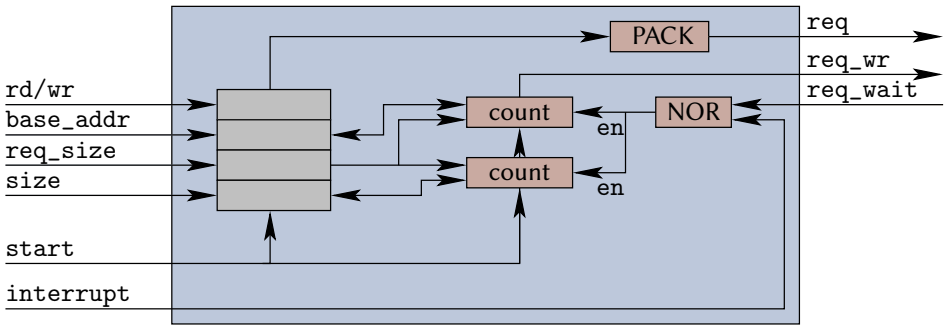


Figure 4.5: Diagramm of the basic request generator core

that is executed once when the request generator is started. '0' represents a read request, '1' represents a write request. So, setting this parameter to "00010001" means "send three read requests, followed by one write request and repeat this sequence once". The second sequence parameter uses the same encoding and is repeatedly executed after the setup phase. The setup phase is useful when data has to be read, processed and written back to memory — in this case, the datapath pipeline gets filled before data is written back.

The third version of the request generator posts a sequence of read and write requests, but does not execute a static sequence. Instead, it monitors the `write` signal of the M2S channel. When a configurable amount of data is sent to this channel, an appropriate write request is inserted into the sequence of read requests. Table 4.3 gives an overview of the resources required by the three different request generator variants on an Altera Stratix II FPGA.

	Implementation		
Resource	basic	sequenced	wr-insert
ALUTs	106	308	118
REGs	66	170	67

Table 4.3: Resources required by the different requester implementations

4.4.4 IMORC-to-Register Interface Core

Cores typically need to be configured with some parameters describing the job to be processed. Examples are start addresses of data to be processed and the volume of this data. These parameters are packed into a job message that needs to be decoded by the core. The IMORC-to-Register interface core (I2R-IF) is a utility core that performs this decoding. The core provides a configurable amount of internal registers that can be accessed in two ways: on the one side, an IMORC slave interface is available. Messages appearing on

this interface are decoded and the appropriate read/write operations are executed on the register block. The second interface provides direct read/write access to the registers. Here each register provides a data bus that presents the register's current value to the user logic and a data bus used for writing updated values to the register when a corresponding signal `set` is asserted. Additionally, for clearing the stored value a signal `reset` is available for each register. The precedence of the two ports can be configured before synthesis.

Additionally, one of the registers can be configured to block the IMORC slave interface. If this register holds a non-zero value, the IMORC slave interface is blocked. Read and write requests appearing on this interface are not processed until the register is reset again. The purpose of this feature is to support chaining of multiple jobs. The I2R-IF receives a job message on the IMORC slave interface, decodes it and sets the appropriate registers, including the register that is set to block. The core starts its operations and when finished resets the blocking register. Then, the I2R-IF is able to accept the next job message. Other cores sending jobs to this core do not need to synchronize with the finalization of the job. They can simply send a stream of job messages and, if they need to be informed of the finalization of the last job, send a final read request that will be responded by the I2R-IF after all jobs have been processed.

Figure 4.6 shows a sample core using the IMORC-to-Register interface core. A master core can send a job description to the I2R-IF. The register interface can be connected to one or multiple request generator cores directly, providing a straightforward method for generating the control unit of a core. Table 4.4 summarizes the resource consumption of the I2R-IF in three different configurations on an Altera Stratix II FPGA.

	# Job Registers		
Resource	5	10	15
ALUTs	104	179	211
REGs	179	340	500

Table 4.4: Resource usage of the I2R-IF in different configurations (Register width: 32bit)

4.4.5 Register-to-IMORC Interface Core

Corresponding to the I2R-IF presented in the previous section, IMORC also provides a Register-to-IMORC interface core (R2I-IF). The behaviour is similar to the I2R-IF, but this time the IMORC interface of the core is an IMORC master interface. While the I2R-IF is used for decoding job messages and forwarding the information to compute cores, the R2I-IF is used for encoding such job messages. Figure 4.7 shows the block diagram of the R2I-IF.

The following signals are provided on the user interface of the core:

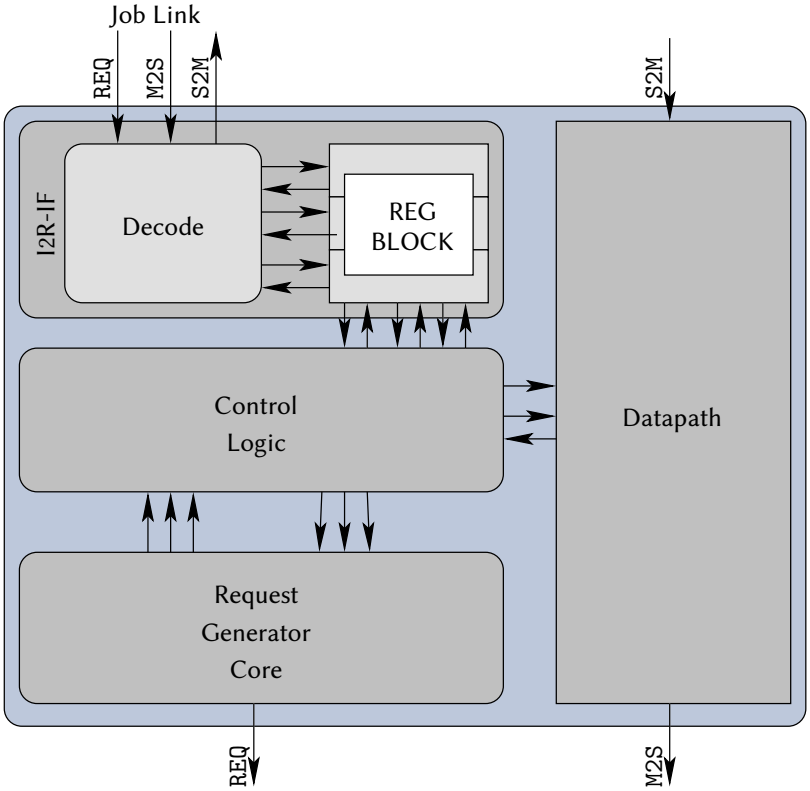


Figure 4.6: Sample core using the IMORC-to-Register interface core

- `send`: when asserted, values provided on `regs_i[0..n-1]` are packed into an IMORC packet and transmitted to the slave core
- `rd_all`: when asserted, a read request is transmitted to the slave core for getting all registers' values, the values responded are shown on `regs_o[0..n-1]`
- `rd_single`: read value of a single register, resulting value is shown on `regs_o[rd_id]`
- `rd_id`: id of the register that is read
- `busy`: communication is ongoing, no read/write requests will be accepted
- `regs_i[0..n-1]`: input values for the registers that have to be sent to the slave core
- `regs_o[0..n-1]`: value of the registers as read from the slave core

Using this interface, a core can send jobs to another core by setting the appropriate job parameters on `regs_i` and asserting the signal `send`. The CTRL block instructs the request generator to send a write request onto the REQ channel. The data provided on `regs_i` ist stored in the embedded register block (by asserting `wr_all`) and successively sent to the M2S channel. During this phase, the signal `busy` is kept asserted, so the user logic is not allowed to send further jobs. When `busy` is deasserted, the interface is ready

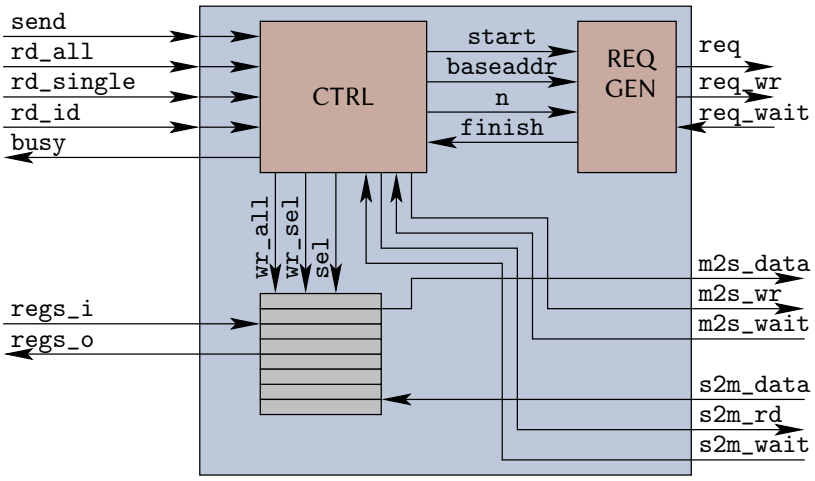


Figure 4.7: Block diagram of the Register-to-IMORC interface core

to accept further jobs.

In order to synchronize with the completion of a job, the master core has to perform a read to one or all of the slave's registers. The user logic asserts `rd_all` for reading all registers or sets `rd_id` to the ID of the desired register and asserts `rd_single` for reading only the specific register. Again, a corresponding read request is sent to the REQ channel. The updated register values are read on the S2M channel and stored in the embedded register block. If the accessed slave interface contains a blocking register, no answer will arrive until this register is deasserted. In this case `busy` stays asserted until the complete read request is processed, i. e., the embedded register block contains the updated values which are presented to the user logic on `regs_o`. Table 4.5 summarizes the resource consumption of the R2I-IF for three different configurations on an Altera Stratix II FPGA.

	# Job Registers		
Resource	5	10	15
ALUTs	136	210	253
REGs	229	390	550

Table 4.5: Resource usage of the R2I-IF for different numbers of job registers (Register width: 32bit)

4.4.6 Farming Cores

Farming is a method often used in parallel computer programming. It means that data is distributed between multiple processors and each one autonomously operates on its own set of data. IMORC also supports this programming model by providing a dedicated job scheduler - the farming core. Configuration parameters of the farming core are the format of the job description (number and size of registers) and the number of worker cores. The farming core provides an IMORC slave and an IMORC master port. The master port is connected to the compute cores' links using the $1 : n$ connection scheme described before. On its slave port the farming core receives job messages which are then distributed to the compute cores using round robin. Figure 4.8 shows the block diagram of a farming core configured for managing three worker cores. When a new job request is sent to the slave port, the Select Worker module forwards it to the next worker core. Additionally, it decodes the request and forwards the amount of data that has to be read or written (signal `n` in Fig. 4.8) to M2S or S2M datapath modules along with the port number to send the data to or to read the data from (signal `p`).

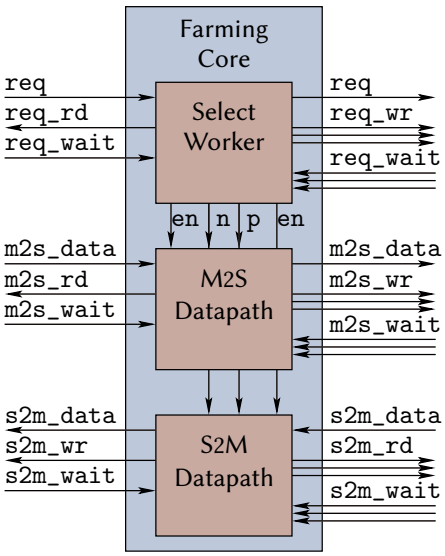


Figure 4.8: Block diagram of the farming core managing three worker cores

A very important parameter if using the farming core is the depth of the FIFOs used in the links that connect the farming core to the worker cores. Deep FIFOs in these links will introduce a rather large overhead in terms of area consumption. Additionally, if these links are able to buffer multiple job requests, the performance of the overall system may be degraded due to an inefficient job scheduling. This is of importance in particular if the jobs largely differ in their expected runtime. The worst case would be that one core is assigned with several long-running jobs while others can quickly process their short-running jobs

and then become idle. So, the IMORC links connecting the worker cores to the farming core usually should be configured to buffer exactly one complete job message. The S2M channels also usually do not need to buffer a large amount of data and can be configured with a FIFO size of one element. Furthermore, the farming core's implementation is rather simple and thus should be able to operate in the clock domain of the worker cores. This enables an implementation as synchronous FIFOs that will use registers for small FIFOs. The bitwidth conversion modules usually can also be omitted. With these properties, such IMORC links will be implemented with a very small area consumption. Clock domain conversion and further buffering of the job descriptions can be performed in the link which connects the core generating the jobs to the farming core.

Additionally to job messages, the farming core can respond to status requests, which are implemented as read requests. The read request is forwarded to all worker cores, which in turn will respond to the request as soon as the jobs already scheduled completed processing. When all worker cores have responded to this request, the farming core will respond to the original read request, informing the requesting core that all jobs scheduled before the status request are finished. Table 4.6 shows the resource usage of the farming core for 5 and 10 compute cores on an Altera Stratix II FPGA.

	#,Worker Cores	
Resource	5	10
ALUTs	105	139
REGs	27	42

Table 4.6: Resource usage of the farming core (5 × 32 bit wide job registers)

4.5 IMORC on the XtremeData XD1000

The XtremeData XD1000 is a dual socket Opteron system whose sockets are equipped with a 2.2 GHz AMD Opteron processor and a module featuring an Altera Stratix II EP2S180-3 FPGA. Since the Opteron is a NUMA style architecture, each processor socket is connected to a dedicated set of memory modules. The XD1000 system comes with 4 GB of main memory (DDR SDRAM) for each the Opteron and the FPGA. CPU and FPGA communicate via a HyperTransport link which provides a rather low latency communication. Physically, the FPGA is connected to the processor using a 16 bit HyperTransport link with 800 MT/s (MegaTransfers per second).

4.5.1 The FPGA

The XD1000 is equipped with the largest Altera Stratix II device available in 2006, the EP2S180-3. This section summarizes the resources available in this FPGA. Since an in-depth discussion of the FPGA is out of scope of this work, for further information refer to the device handbook [31]

Logic Resources

The Stratix II FPGA consists of a two-dimensional architecture to implement custom logic. Logic resources are grouped into logic array blocks (LABs), each of which consists of eight adaptive logic modules (ALMs). An ALM is the basic building block of custom logic in the Stratix II devices. Figure 4.9 shows the block diagram of an ALM. It consists of a number of look-up tables (LUTs) based resources that can be divided into two adaptive LUTs (ALUTs). Precisely, each ALUT consists of one four-input LUT and two three-input LUTs. Each ALM provides a total of eight inputs for the ALUTs. This way, several functions can be implemented by an ALM:

- Each ALM can implement two arbitrary functions with four inputs,
- two arbitrary functions with three and five inputs,
- one arbitrary function with up to six inputs, and
- certain functions with seven inputs.

Additionally, each ALM contains two adders with carry chains, multiplexers and registers with a register chain. The register chain can be used for implementing shift registers.

Memory

The Stratix II FPGA provides three different types of on-chip memory: M512, M4K and M-RAM. M512 is the smallest RAM block containing 576 bits and supports different configurations from 512 × 1 to 32 × 18. M4K contains 6 408 bits in configurations from $4\,K \times 1$ to 128 × 36 and M-RAM contains 589 824 bits in configurations from $64\,K \times 1$ to $4\,K \times 144$. The memories can be operated in different modes, such as single-port memory, simple or true dual-port memory, ROM, shift register, etc. Not all memories provide support for all operation modes. A detailed description of the memories can be found in the literature [31]

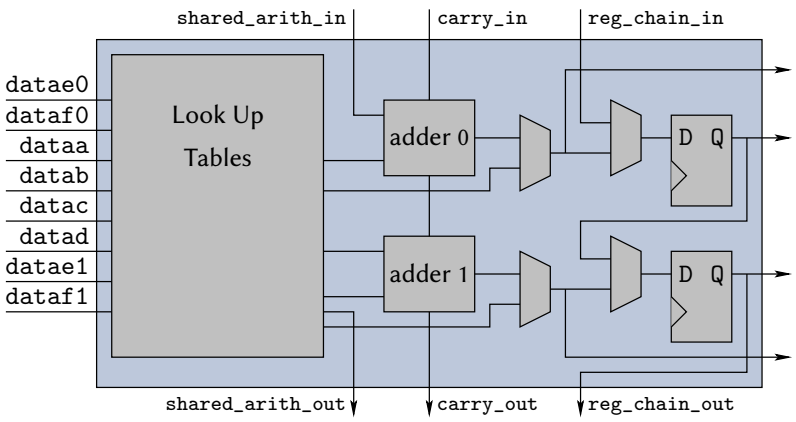


Figure 4.9: Diagram of an ALM in the Altera Stratix II FPGA

Digital Signal Processing Blocks

To support Digital Signal Processing (DSP) functions, the Altera Stratix II EP2S180 FPGA provides 96 dedicated DSP blocks. Each DSP block can be configured to support up to eight 9×9 bit multipliers, four 18×18 bit multipliers or one 36×36 bit multiplier. Additionally, the DSP blocks contain an adder output block that can be configured for adding, subtracting or accumulating the results of the multipliers, forming a multiply-add or multiply-accumulate function.

4.5.2 Host Interface

HyperTransport (HT) is an open standard defined by the HyperTransport Technology Consortium [14]. Devices are connected by a point-to-point link that is implemented using two unidirectional sets of signals:

- `CAD` (Command, Addresses and Data) is a between 2 bits and 32 bits wide signal and transports the packets,
- `CTL` is a one bit wide signal that is asserted when `CAD` carries a control packet and
- `CLK` is a 1, 2 or 4 bits wide clock signal — each byte of `CAD` has a separate clock

HyperTransport is a packet-based protocol. Each packet consists of a header and can have data attached. Packets are grouped into three virtual channels: Posted for packets that do not expect a response, Non-Posted for packets requiring a response and Response for responses to packets received on the Non-Posted channel. Packets are basically separated into control and data packets. Figure 4.10 shows the basic structure of a read/write request packet. The purpose of the fields is as follows:

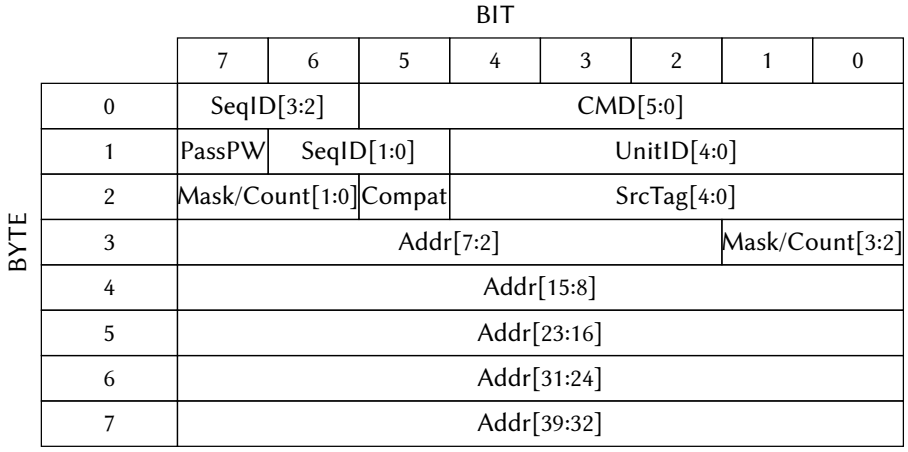


Figure 4.10: Structure of a HT read/write request packet

- CMD: command field, e. g., byte or doubleword read or write request
- SeqID: used for grouping requests. All requests with the same SeqID have to be strongly ordered within a virtual channel. Requests with SeqID 0x0 are not part of a sequence, thus no sequence-ordering is required for such requests.
- PassPW: packets with this bit set may pass packets in the posted request channel.
- SrcTag: tag used to identify all transactions, ie, to match responses with their requests.
- Addr: doubleword address accessed by the request.
- Mask/Count: For doubleword requests, indicates the number of doubleword data elements to be transferred. For byte requests, exactly one doubleword is transferred in the data packet and this field is used for masking out non-relevant bytes.

Read response packets are only 4 bytes long since they do not need to provide an address. Data packets directly follow a write request/read response packet and contain the corresponding amount of data. Several other packet types exist, like broadcast, atomic read-modify-write etc.. For a detailed reference of the HT protocol refer to the specification [69] or to the HT System Architecture book [34].

For communication between host CPU/memory and FPGA, an IMORC interface core to HyperTransport exists which is based on the HT cave presented in [110]. During system boot, the host CPU and the cave negotiate the HT link parameters (i. e., the link's clock and width). Then the host reads the cave's capabilities and configures it. The cave maps up to three distinct address regions into the address space of the host's CPU.

After this initialization, accesses to this address space are translated into HT read or write request packets. Writes into one of these address regions appear as a write packet on the incoming posted channel, reads appear as a read packet on the non-posted channel. The cave performs some decoding (e. g., the address region that was assessed by an incoming packet) and forwards it to the user logic. Connection to user logic is separated into 6 interfaces, one transmitting (Tx) and one receiving (Rx) interface for each virtual channel (P - Posted, NP - Non-Posted, R - Response).

Figure 4.11 displays the connection of the HT cave to the corresponding IMORC interface. The IMORC interface for incoming transfers converts packets that hit one of the first two address regions into equivalent IMORC packets which are posted on two separate IMORC links. For write transfers this conversion is straightforward — the corresponding IMORC link is taken from the address region ID. Address and size are extracted from the corresponding fields in the HT packet and the appropriate amount of data is forwarded from the cave's TxP data FIFO to the corresponding IMORC link's M2S channel. For reads, the decoding is the same regarding the packet header. However, additional data has to be forwarded to the HT's response channel to be able to generate a response packet header and to know from which IMORC link the data has to be read. Precisely, this data involves the TAG, SIZE and the address region. This data is forwarded using a synchronous FIFO. The response datapath generates an appropriate HT packet header out of these information, sends it to the corresponding FIFO interface and forwards the appropriate amount of data from the corresponding IMORC link's S2M channel to the HT cave.

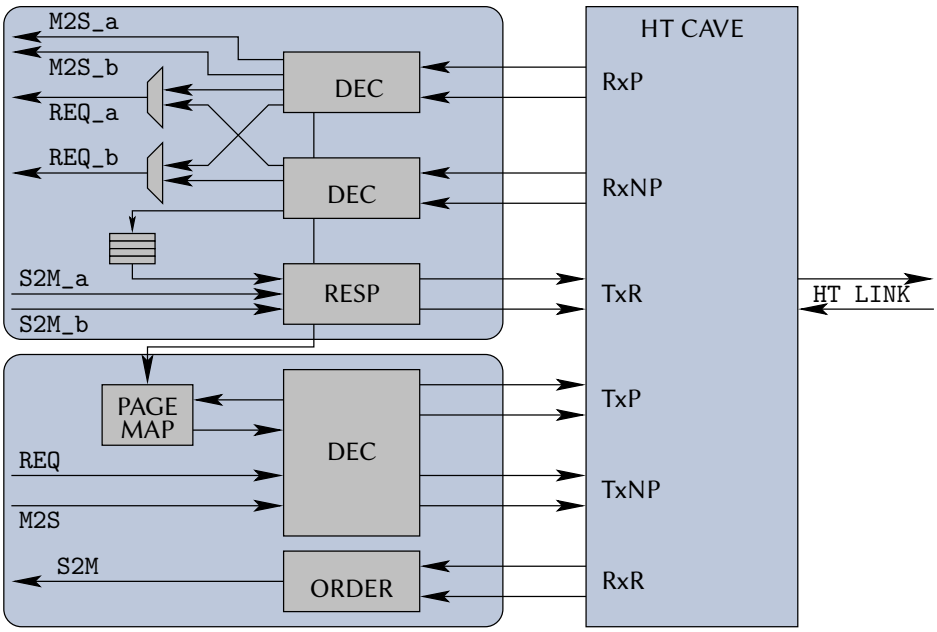


Figure 4.11: Block diagram of the HyperTransport interface core

The third address region is used for supporting communication into the other direction. The FPGA may send packets to the host CPU for directly accessing its local memory. However, since the Opteron CPU uses virtual addresses, it is not sufficient to inform compute cores running on the FPGA at which base address the desired data begins. The third address region accesses an embedded block of memory for mapping a set of physical page addresses of the CPU into a continuous address space on the FPGA. IMORC packets arriving at a separate IMORC slave interface are translated into corresponding HT packets this way. The upper bits of the IMORC address act as a pointer into the page mapping table, the lower bits form an offset to the page. For read requests, an additional counter is used for generating the TAG field of the HT request. HT supports out-of-order responses, while IMORC expects responses to arrive in-order. To overcome this issue, the receiving response datapath first stores incoming data packets into a local memory. This local memory contains one block large enough for storing a full-sized HT data packet per possible TAG. Data is written to the block corresponding to the TAG field of the HT header and marked as valid. Starting with tag 0, the valid flag of the next tag which has to be processed is monitored. In the case the block is marked as containing valid data, the data is forwarded to the S2M channel of the corresponding IMORC link and the block is marked as invalid.

Additionally to memory access, the IMORC slave link can be used for sending interrupt messages to the host. Writes to a configurable address occurring at the slave IMORC link are translated into HT interrupt packets and sent to the posted channel of the HT cave. Table 4.7 lists the resource requirements for the HT host interface core.

Resource	# used	% of Stratix II EP2S180
ALUTs	8791	6.125 %
REGs	5209	3.629 %
M512	1	0.108 %
M4k	57	7.422 %
M-RAM	1	11.111 %

Table 4.7: Resource requirements for the HT host interface core

4.5.3 External DDR Memory Access

For off-chip DDR memory access IMORC wraps the Altera DDR SDRAM controller core, which can access memory in blocks of configurable burst sizes. The DDR SDRAM provided by the XD1000 system is a 128 bit wide memory, which can be accessed in bursts of 2, 4 or 8 clock cycles. The Altera controller can be configured for the maximum burst size to be used.

The user interface signals of the SDRAM controller can be grouped into three classes: control, read data and write data. Figure 4.12 gives an overview of these signals. The

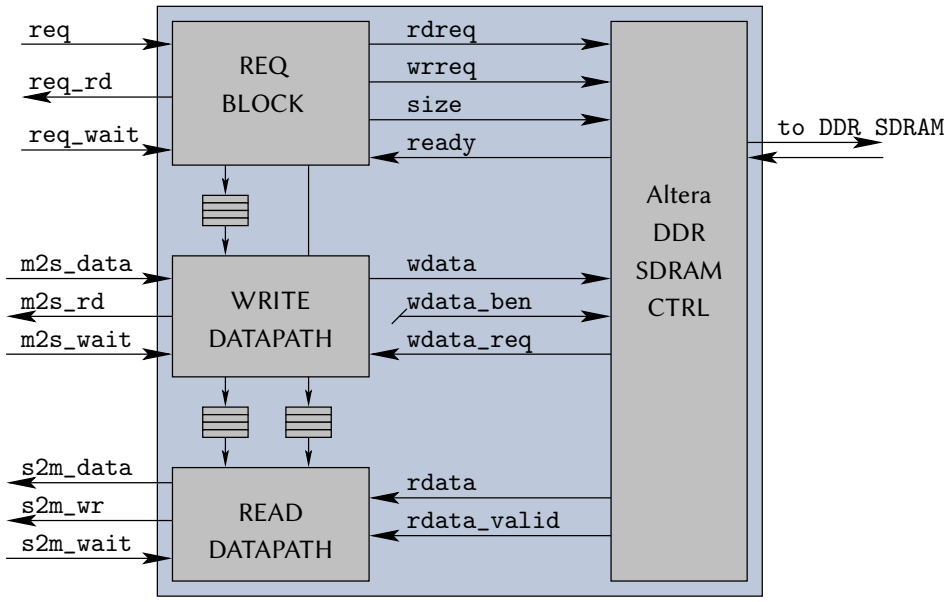


Figure 4.12: Block diagram of the XD1000 DDR SDRAM interface core

control signals are used for instructing the controller to perform a burst read or write. The `addr` signal's granularity is 256 bit, the `size` defines the number of 256 bit words which have to be read or written. Multiple requests can be sent in a sequence, as long as the `ready` signal is asserted.

The write datapath calculates an error correcting code (ECC) for the data; ECC code and data are then forwarded to the controller when the `wrdata_req` signal is asserted. The read datapath checks the ECC, performs error correction if necessary and forwards the data to the S2M channel of the IMORC link as soon as `rddata_valid` is asserted.

DDR SDRAM transfers usually cover complete bursts, i. e., 2, 4 or 8 clock cycles in the DDR clock domain or 1, 2 or 4 clock cycles in the SDR domain. Hence, depending on the burst size configuration of the controller, 256 bit, 512 bit or 1024 bit are usually read or written during each transfer. For reads, the controller can additionally break the burst after 2 or 4 cycles in the DDR clock domain and only responds the number of 256 bit words requested by the `size` signal. Write transfers always have to cover a complete burst. Data can usually be masked out using the `wdata_ben` signal on the user interface, which is forwarded to the memory using the `dm` signal. However, since the `dm` signal is not connected to the memory in the XD1000, complete bursts of 256 bit, 512 bit or 1024 bit are written to the memory. To also support subword writes, the IMORC interface implements a read-modify-write cycle: data is read from the target position of the memory first, then forwarded to the write datapath using a short FIFO and finally multiplexed onto the data which is written. While this approach certainly introduces some overhead, it ensures that

on-chip or host memory is transparently exchangeable with DDR SDRAM on the XD1000. Table 4.8 lists the resource requirements of the DDR SDRAM.

Resource	#used	% of Stratix II EP2S180
ALUTs	1992	1.380 %
REGs	2159	1.504 %
M4k	8	1.042 %

Table 4.8: Resource requirements for the DDR CTRL core

4.6 IMORC Infrastructure Cores and Accelerator Generation

Figure 4.13 outlines the generation of accelerators based on the architecture model. The different steps in this flow are supported by a code generation tool implemented in the language Ruby.

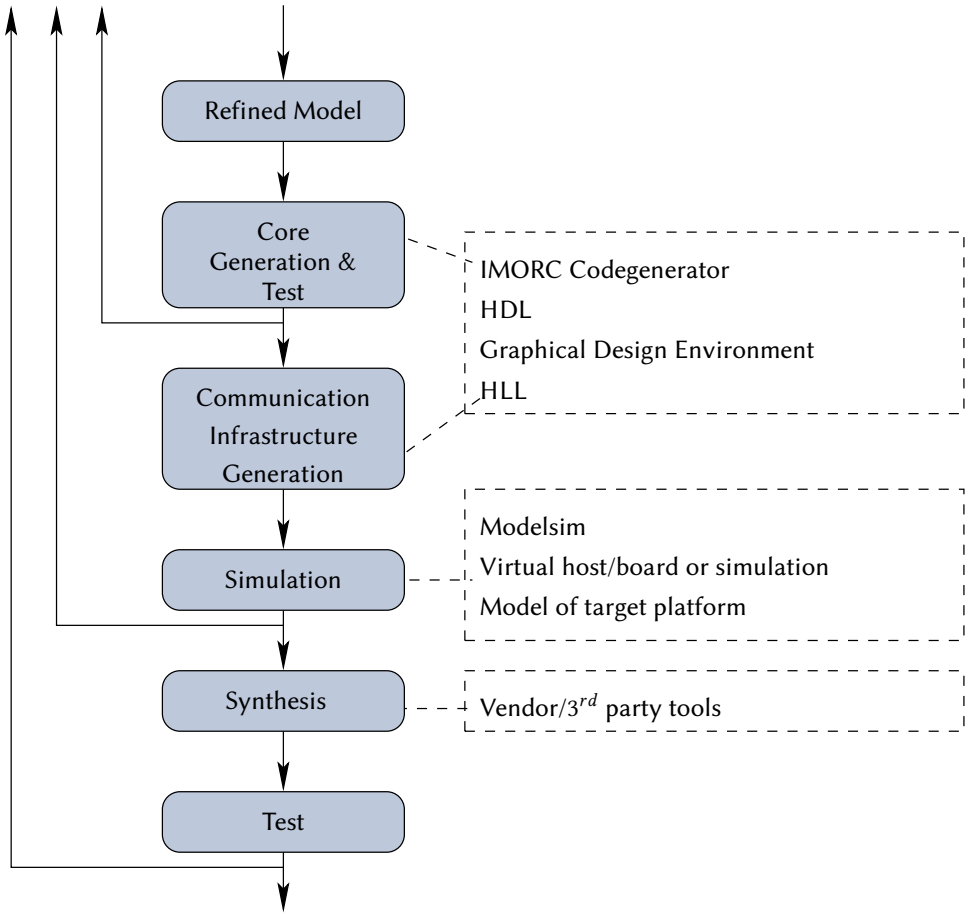


Figure 4.13: Architecture generation flow diagram

4.6.1 Core Generation

Core generation typically begins with specifying the interface to the core, e. g., how many IMORC slave and master ports it provides and the width of the different S2M and M2S channels. In the next step several supporting cores such as a I2R-IF, request generator cores etc. are instantiated, and custom functionalities are implemented.

Listing 4.1 shows the entity definition of an IMORC core with two IMORC ports: one master and one slave port. While the whole process of core generation can be performed using VHDL, Verilog or some kind of graphical design tools, the ruby-based code generator can simplify this process in some ways. Listing 4.2 shows the corresponding code generator script which produces the entity definition in Listing 4.1.

Additionally, the generator generates the architecture implementation of the cores. For this purpose, the designer can define signals, connect signals and IMORC links and instantiate variants of the utility cores and other cores generated by the code generator. Listing 4.3 shows the exemplary code of a custom core. The core definition myCore is implemented as a subclass of class Core. The width of the datapath and the number of job registers are parameterized and need to be defined when instantiating the class. The constructor adds the ports rst and clk, a 32 bit wide IMORC slave port s and an IMORC master port m that is as wide as the datapath. It then generates an instance of the I2R-IF, maps the ports rst, clk and s to the corresponding ports of "myCore". The method genCustomVHDL() can be used for adding arbitrary VHDL code to the architecture definition.

The VHDL for this core can now be generated by generating an instance of class myCore and calling its method genVHDL(), which is defined in its superclass Core. The resulting VHDL file contains the code implementing the I2R-IF and the core myCore, including a component definition for the I2R-IF.

The code generator is also able to insert load sensors for monitoring arbitrary single bit signals. This is done by calling the method loadSensor(< signal >), where < signal > is the name of the signal to observe. This method automatically generates the needed signals for accessing the load sensor. By default, these signals are routed to the external interface of the core. The load sensor signals of instantiated cores are also forwarded to the external interface by default, hence a core has access to every load sensor at a lower level. The load sensors thereby are identified by a unique name that is generated by the name of the signal monitored and the core's instance name. Each time a load sensor is promoted to a higher level in the hierarchy, the name of the core's instance name is prepended to the load sensor's name. Routing the load sensor signals to the external interface can also be suppressed. This is useful if the core itself implements the monitoring core and therefore has to access the sensors.

4.6.2 Communication Infrastructure Generation

The communication infrastructure generation consists of instantiating cores and connecting them using IMORC links and slave arbiters. Using the methods presented above, the IMORC code generator can be used for this job. For this purpose, the top level design entity is regarded as a core. The external interface of this top level core has to match the interface that connects the FPGA to external resources, such as the host interface and

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4  library imorc;
5  use imorc.defs.all;
6  use imorc.tool.all;
7  use imorc.settings.all;
8
9  entity myIMORCcore is
10   generic(
11     S_WIDTH : INTEGER := 32;
12     M_WIDTH : INTEGER := 32
13   );
14   port(
15     rst         : in  std_logic;
16     clk         : in  std_logic;
17
18     -- IMORC slave port --
19     s_clk       : out std_logic;
20     s_req       : in  std_logic_vector(IMORC_REQ_BITS-1 downto 0);
21     s_req_wr    : in  std_logic;
22     s_req_wait  : out std_logic;
23
24     s_m2s_data  : in  std_logic_vector(S_WIDTH-1 downto 0);
25     s_m2s_wr    : in  std_logic;
26     s_m2s_wait  : out std_logic;
27
28     s_s2m_data  : out std_logic_vector(S_WIDTH-1 downto 0);
29     s_s2m_rd    : in  std_logic;
30     s_s2m_wait  : out std_logic;
31
32     -- IMORC master port --
33     m_clk       : out std_logic;
34     m_req       : out std_logic_vector(IMORC_REQ_BITS-1 downto 0);
35     m_req_wr    : out std_logic;
36     m_req_wait  : in  std_logic;
37
38     m_m2s_data  : out std_logic_vector(M_WIDTH-1 downto 0);
39     m_m2s_wr    : out std_logic;
40     m_m2s_wait  : in  std_logic;
41
42     m_s2m_data  : in  std_logic_vector(M_WIDTH-1 downto 0);
43     m_s2m_rd    : out std_logic;
44     m_s2m_wait  : in  std_logic
45   );
46 end entity;
```

Listing 4.1: Sample code specifying the interface of an IMORC core with one IMORC master and one IMORC slave port

```
1 require 'imorc'
2
3 mc=Core.new("myIMORCCore")
4 mc.addInput("rst", 1)
5 mc.addInput("clk", 1)
6 mc.addSlave("s", 32)
7 mc.addMaster("m", 64)
8
9 mc.genVHDL("myIMORCCore.vhd")
```

Listing 4.2: Entity definition using the code generator

external memory. In case of the XD1000, this includes the HyperTransport interface and the interface to external DDR SDRAM. The top level core instantiates the compute cores and infrastructure cores, such as slave arbiters, IMORC links, farming cores and interface cores to the host interface and to external memory. Additionally, it may not forward the load sensor signals on the external interface. Instead, if load sensors shall be monitored, the top level core has to instantiate an appropriate monitoring core.

For the XD1000, a class defining the template of such a top level core is provided. It implements the interface of the XD1000 FPGA, instantiates the HT interface core and optionally instantiates a DDR controller interface core. Additionally, it provides methods for specifying the load sensors that have to be monitored. Typically, a designer will implement a subclass of the XD1000 class that instantiates the IMORC infrastructure components and compute cores. In a generator script, this class is instantiated, the load sensors which have to be monitored are selected and the method genVHDL() is called. This generates a VHDL file containing the complete architecture's implementation. This last step can also be done interactively using the interactive ruby shell. This way the designer can, for example, list the available load sensors in the system before selecting the monitored ones.

Using the code generator is in some ways more flexible than directly implementing VHDL code. For example, VHDL does not support a configurable number of ports for an entity. To implement an IMORC slave arbiter with a configurable number of ports in VHDL, the IMORC inputs have to be specified as two dimensional vectors of signals. VHDL supports two variants of such vectors: vectors of vectors like

```
type tdvec1 is array(NATURAL RANGE <>) of
                std_logic_vector(NATURAL RANGE <>);
```

and real two dimensional vectors such as

```
type tdvec2 is array(NATURAL RANGE <>. NATURAL RANGE<>)
    of std_logic;
```

```
1  class myCore < Core
2    def initialize(dwidth, nregs)
3      super("myCore")
4      @dwidth=dwidth
5      @nregs=nregs
6      addInput("rst", 1)
7      addInput("clk", 1)
8
9      addSlave("s", 32)
10     addMaster("m", dwidth)
11
12     i2r_inst=addInstance("JOB_DECODER", I2RIF.new(nregs))
13     i2r_inst.mapPort("rst", "rst")
14     i2r_inst.mapPort("clk", "clk")
15     i2r_inst.mapImorcPort("s", "s")
16
17     for i in 0..nregs-1
18       addSignal("reg_"+i.to_s(), 32);
19       i2r_inst.mapPort("reg_"+i.to_s(), "reg_"+i.to_s())
20
21       addSignal("reg_"+i.to_s()+"_set", 32);
22       i2r_inst.mapPort("reg_"+i.to_s()+"_set", "reg_"+i.to_s()+"
             _set")
23
24       addSignal("reg_"+i.to_s()+"_rst", 32);
25       i2r_inst.mapPort("reg_"+i.to_s()+"_rst", "reg_"+i.to_s()+"
             _rst")
26     end
27
28     def genCustomVHDL()
29       return "
30        --here, custom VHDL commands may be written
31        --for specifying the architecture
32       "
33     end
34 end
```

Listing 4.3: Sample instantiating another core

At design time of IMORC the first option was not practically usable because synthesis tools required at minimum one dimension to be specified during type definition. Since for an IMORC slave arbiter the number of ports and the width of the datapaths may differ for every arbiter that is instantiated, both dimensions have to be parameterizable. The second option can be used, but signal assignment in VHDL becomes complicated, since it is not possible to select a range of signals:

```
signal a : tdvec2(0 to 31, 31 downto 0);
signal b : std_logic_vector(31 downto 0);
...
b <= a(2, 31 downto 0); -- Wrong!

b(31) <= a(2,31);       -- Correct!
```

Using the code generator, custom slave arbiters can be generated with an arbitrary number of IMORC ports that can be accessed directly.

4.6.3 Simulation

For supporting the simulation of the generated system, the system has to be connected to external memories and to a host. A basic memory model is implemented that may be used for simulating the external memory. Additionally to the internal memory core, this model supports the addition of a certain latency to the memory blocks, but does not consider the times needed for loading a page, refreshing etc. as in real SDRAM. Thus, timings may vary between such a simulation and a simulation for the real target platform.

For the host interface, a template exists that provides two IMORC master and one slave port. The slave port accesses an internal memory, simulating host memory. The communication performed on the master ports has to be specified by the designer in VHDL.

Alternatively, if the system targets a specific platform, appropriate external memory controllers and a host interface core for that platform can be instantiated. Simulation then can utilize the platform specific simulation models. In case of the XD1000, a model for the complete accelerator board exists. A HyperTransport Bus Functional Model is attached to the board model which initializes the FPGA just like a real system would do and provides the possibility to send commands to the HT link (e. g., to the FPGA).

Additionally, a load sensor monitor model exists which may be used to observe the sensors available in the system. This model reads out the sensors' data in a configurable interval and writes the values into a configurable file for further analysis.

4.6.4 Synthesis

The IMORC architectural template is completely written in synthesizable VHDL, making IMORC a vendor neutral tool. Specifically, the FIFOs are implemented based on the techniques described in [46] without using any architecture specific components. For generating the full and empty flags of the FIFOs, the read and write pointers need to be compared. Since these pointers reside in different clock domains, grey counters are used for generating these pointers, making an asynchronous comparison reliable. The only vendor-specific components are the system specific host interconnects and memory controllers as well as possibly the on-chip memory blocks if subword writes are needed (see section 4.4.2). These properties ensure that the generic parts of an IMORC system can typically be synthesized by all major vendors' synthesis tools.

4.6.5 Execution and Runtime Monitoring

In the next step the final bitstream can be downloaded to and tested on the target FPGA. The included load sensors can be monitored with the provided monitoring tools. For the XD1000, load sensors can be monitored using the HT host interface or the JTAG interface. When using the host interface, data can be written by the monitoring core into a dedicated memory region of the host processor. The host application has to take care of storing the data for later analysis. The JTAG methodology is suitable for all Altera devices, data can be received using the TCL interface of the Altera toolsuite.

4.7 Chapter Summary

The IMORC architectural template provides a versatile method for implementing different kinds of accelerators. The infrastructure in several ways abstracts from the target hardware platform, such as bitwidth and type of memories. The main focus of the IMORC architecture template is to facilitate the development of accelerators and to gain high performance by maximizing the utilization of the communication channels available. Control and datapaths are separated in IMORC enabling designers to use different implementation methods for these jobs. Helper cores are provided for various commonly used tasks.

While this chapter provided an insight into the assumptions made and the design decisions taken during development of the IMORC architectural template, the benefits still have to be proven. Hence, the next chapters analyze the IMORC architectural template running on a concrete system equipped with reconfigurable hardware — the XtremeData XD1000. The next chapter provides a detailed characterization of this platform which is performed using the IMORC architectural template. Based on this characterization Chapter 6 discusses several case studies on different kinds of accelerators.

5 Architecture Characterization

Mapping task graphs to an existing platform requires numerous parameters of the platform to be taken into account. Especially the communication performance to different kinds of memories and between host and accelerator has to be considered for achieving optimal results. While vendors usually provide raw performance values based on parameters like clock frequencies and data widths, these values are most likely not achievable in real applications due to overheads. Special care has to be taken regarding the measurement: while vendors often provide performance values in GB/s, with 1 GB = 10^9 Byte, in engineering traditionally 1 GB is expected to be 2^{30} Bytes, which is nowadays called a *GibiByte* (GiB). Additionally, performance values usually depend on the actual communication scheme. For example, if accessing off-chip memory, the maximum performance can usually only be achieved when accessing consecutive addresses and always performing full-burst transfers. Such conditions cannot always be achieved in real applications. For enabling designers to find an ideal mapping of memory tasks to actual memory resources on a real platform and for a first performance estimation, IMORC provides a versatile benchmarking infrastructure for gathering such performance values based on configurable parameters. This chapter introduces the benchmarking infrastructure and presents an architecture characterization of the XtremeData XD1000 reconfigurable platform, which also was the target platform for the case studies presented in Chapter 6.

5.1 The IMORC Benchmarking Infrastructure

The IMORC benchmarking infrastructure consists of one or multiple benchmarking cores, which communicate with the target memory to be characterized using the IMORC infrastructure.

5.1.1 The Benchmarking Core

The benchmarking core is used to measure the performance achieved on one single IMORC link. Its interface consists of two separate IMORC ports — a slave port used for setting the benchmark's parameters and a master port for accessing the target memory. The slave port connects an I2R-IF used for decoding the current benchmark's parameters. The decoded parameters are forwarded to a control core that configures an IMORC request generator core and the datapath.

Table 5.1 summarizes the runtime parameters of the benchmarking core. The base address is the first request address to be sent by the core. The request size is the size of each request to be performed. After a request is submitted, the address increment value is added to the last request address, allowing to not only benchmark subsequent accesses to the memory but also strided accesses. The benchmark type defines whether the read or the write bandwidth should be measured. Alternatively, it can be set to pattern allowing an arbitrary sequence of read/write requests to be submitted, depending on the value of the pattern parameter. The pattern parameter is a bitvector containing a '1' if a write is to be performed and a '0' for reads.

Parameter	Description
base address	starting address of the benchmark
request size	size of each request
address increment	value by which the destination address for each transfer increments
benchmark type	type of the benchmark, can be read, write or pattern
pattern	when benchmark type is pattern, this parameter defines the access pattern

Table 5.1: Runtime parameters of the benchmarking core

An additional register in the I2R block is used for starting the core. Depending on the configuration parameters, the request generator starts posting requests to the REQ channel. Additionally, the number of transfers that have to occur on the M2S and S2M channel are calculated and forwarded to the datapath. Here, a counter is initialized with this number and decremented whenever a data read or write to the channel occurs, i. e. if the corresponding `wait` signal is deasserted. Data read from the S2M channel is ignored; the data written to the M2S channel consists of pseudo-random values. When the counters become zero, the benchmark is finished and the start register of the I2R block is reset for

unblocking the corresponding IMORC link. A separate counter is used for counting the clock cycles needed. The value is written to another register of the I2R block and can be read by the host. Alternatively, the benchmarking core can be configured for using an IMORC load sensor for counting the clock cycles and for providing the host with this information.

The data width of the M2S and S2M channels is a configurable parameter to be set before synthesis. This way, not only the maximum performance achievable by the memory can be measured but also the maximum performance of a core with a specific data width.

5.1.2 Contention Benchmarking

In real accelerators it is often necessary to map multiple memory tasks to the same memory core, or to have multiple compute cores accessing the same memory task. Simultaneous access to the same memory can be implemented in IMORC by using the slave arbiters, as presented in Chapter 4. However, if multiple cores access the same memory using different access schemes, the actual performance of the accelerator is hard to predict even if detailed information about the communication performance for different schemes is available.

For measuring the impact of multiple cores accessing the same memory, the IMORC benchmarking infrastructure integrates a configurable number of benchmarking cores into a complete system. The cores are connected to the target memory using a slave arbiter. An additional controller is responsible for forwarding benchmarking parameters to the cores, to monitor the finish condition and to forward read requests for getting the value of the cycle counters. Each core can be configured with different parameters for simulating multiple cores each performing a different access scheme.

5.2 Performance Characterization of the XD1000

Figure 5.1 pictures the memory layout of the XD1000 and the communication channels as introduced in Section 4.5. First, the FPGA can be the target of communication events initiated by the CPU, depicted in the figure by labels B_{wr}(HT) and B_{rd}(HT). Second, the FPGA can directly access on-chip memory ($B_{wr/rd}$(IM)), off-chip DDR SDRAM ($B_{wr/rd}$(EM)) and host memory ($B_{wr/rd}$(HM)).

Internal memory is configurable regarding the bitwidth, the memory depth and the clock rate in wide ranges. Since this memory is accessible with a latency of only one clock cycle, the performance achievable can be directly calculated based on these parameters. Hence, the further characterization of memory access performance focuses on host memory and on DDR SDRAM.

Table 5.2 summarizes the capacities and the theoretical maximum bandwidths for all memories in the XD1000 system, as taken from specifications and data sheets. Note that

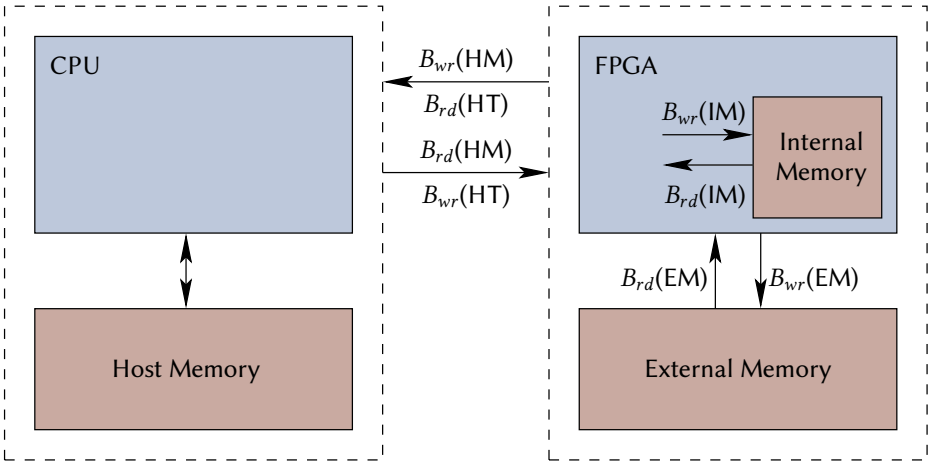


Figure 5.1: Memory architecture of the XD1000 architecture

the HT link is implemented as two unidirectional links, each one providing a theoretical bandwidth of 1.6 GB/s, resulting in a theoretical bidirectional peak bandwidth of 3.2 GB/s. The DDR SDRAM on the other hand is connected using one bidirectional bus, which is shared for read and write access. As a consequence, the 5.4 GB/s mentioned are a theoretical maximum for read only, write only and also combined read/write accesses. The values for CPU initiated transfers are not presented in the table since the theoretical parameters do not differ from the values of host memory access initiated by the FPGA. The only difference is that such transfers produce some overhead on the host CPU.

Memory	Capacity	Parameter	Bandwidth
Host	4 GB	$B_{rd}(HM)$	1.6 GB/s
		$B_{wr}(HM)$	1.6 GB/s
External	4 GB	$B_{rd}(EM)$	5.4 GB/s
		$B_{wr}(EM)$	5.4 GB/s
Internal	1 MB	$B_{rd}(IM)$	$\gg$
		$B_{wr}(IM)$	$\gg$

Table 5.2: Capacities and theoretical bandwidths for the XD1000

The bandwidth figures in Table 5.2 can be considered as upper bounds. Any concrete implementation such as IMORC will possibly not be able to meet these theoretical figures due to controller and protocol overheads. Typically, such overheads make the bandwidth dependent on the request size. Furthermore, the bandwidth achieved in an implemented accelerator can be reduced by contention, in case that several cores compete for accessing the host or external memory simultaneously.

5.2.1 CPU ↔ Host Memory Bandwidth

The bandwidth between CPU and host memory is characterized using the RAMspeed [22] benchmark suite with different parameters. RAMspeed performs different kinds of operations to measure the read and write performance separately as well as the combined performance of the system's memory. RAMspeed consists of two kinds of benchmarks, called *mem* and *mark* benchmarks.

The mem benchmarks (INTmem, FLOATmem, MMXmem and SSEmem) perform synthetic operations on an array of data to measure the memory performance with realistic workload. In detail the operations are:

- Copy transfers data from one location to another ($A = B$)
- Scale modifies the data before writing it to the target location by multiplying it with a constant value ($A = m \cdot B$)
- Add reads data from two memory locations, adds them and writes them back to a third memory location ($A = B + C$)
- Triad is a combination of scale and add — it reads data from two locations, scales data from the first location and adds the result to the value read from the second location before writing the result back ($A = m \cdot B + C$)

The benchmarks sum up the amount of data accessed by each operation and with the resulting value calculate the throughput through the corresponding execution unit. RAMspeed additionally provides variants of the MMX and SSE benchmarks that insert explicit prefetching instructions into the code. While the integer and floating point benchmarks are implemented in C, the MMX and SSE benchmarks are implemented in machine specific assembler code.

Figure 5.2 shows the results of the RAMspeed mem benchmarks on the XD1000. The bandwidth achieved is between 2.1 GiB/s and 2.5 GiB/s for the Integer benchmarks and between 2.1 GiB/s and 2.7 GiB/s for the Float benchmarks. The average bandwidths achieved are about 2.34 GiB/s and 2.46 GiB/s, respectively. These values are much lower than the peak values theoretically achievable by the memories. The figure also shows that the utilization of the MMX and SSE units do not directly exert an influence on the bandwidth achieved — the performances measured with these units are very close to the integer and floating point benchmarks. A significant improvement is produced by inserting data prefetching instructions into the computational loops. In this case, the performance about doubles, resulting in values of between 3.9 GiB/s and 4.7 GiB/s for the MMX benchmark and between 3.9 GiB/s and 4.8 GiB/s for the SSE benchmark, respectively.

The benchmarks also reveal another interesting fact: for the benchmarks without manual prefetching, the memory bandwidth slightly increases with the complexity of the operations performed (Triad > Add > Scale > Copy). The benchmarks with prefetching code inserted showed the contrary behavior — with an increased complexity of the operations performed, the resulting bandwidth dropped in this case.

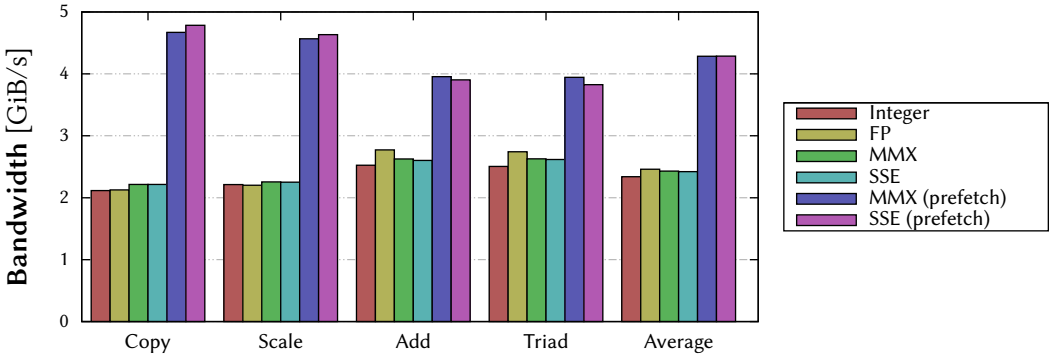


Figure 5.2: Results of the RAMspeed benchmark on the XD1000

The RAMspeed mark benchmarks on the other hand measure the bandwidth achievable when accessing memory with different block sizes. The benchmark hereby consists of a scalar variable *a* and a memory block *mem*. For the read benchmarks, the values stored in *mem* are successively copied to variable *a*, reading each element of *mem* exactly one time. For the write benchmark, value *a* is exactly written one time to each location of *mem*. If the size of the memory block accessed fits into the L1 or L2 cache, most of the copy operations can be typically executed directly in this cache without direct access to main memory. This way, the mark benchmarks measure not only the memory performance but also the performance achievable when accessing the different levels in the cache hierarchy. Figure 5.3 presents the results of this benchmark on the XD1000.

Basically, all benchmarks except the MMX and SSE write benchmarks with manual prefetching behave similarly. Transfers that can be completely performed in L1-cache (64 byte on the Opteron 248 processor) perform best, achieving from 16 GiB/s to 32 GiB/s depending on the operation and datatype. For the L2-cache, the bandwidths achieved lie much closer together. Interestingly, MMX and SSE writes with prefetching however do not benefit from L1-cache or L2-cache at all. The throughput in these benchmarks constantly is around 6 GiB/s for all block sizes. This value is much higher than the bandwidth achieved by the other benchmarks when accessing main memory, but lower than access to L1- or L2-cache by the other benchmarks.

5.2.2 CPU ↔ FPGA Communication Initiated by the CPU

For characterizing $B_{wr/rd}$(HT) the benchmarking application on the CPU maps the FPGA into its address space and performs `memcpy` operations with different sizes to/from this address space. The upper bound for these bandwidth values are given as 1.6 GB/s per direction, which are however physical peak values. The HT link in each direction is 16 bits wide and running at 400 MHz DDR, resulting at a maximum of 800 MT/s or 800 M/s ×

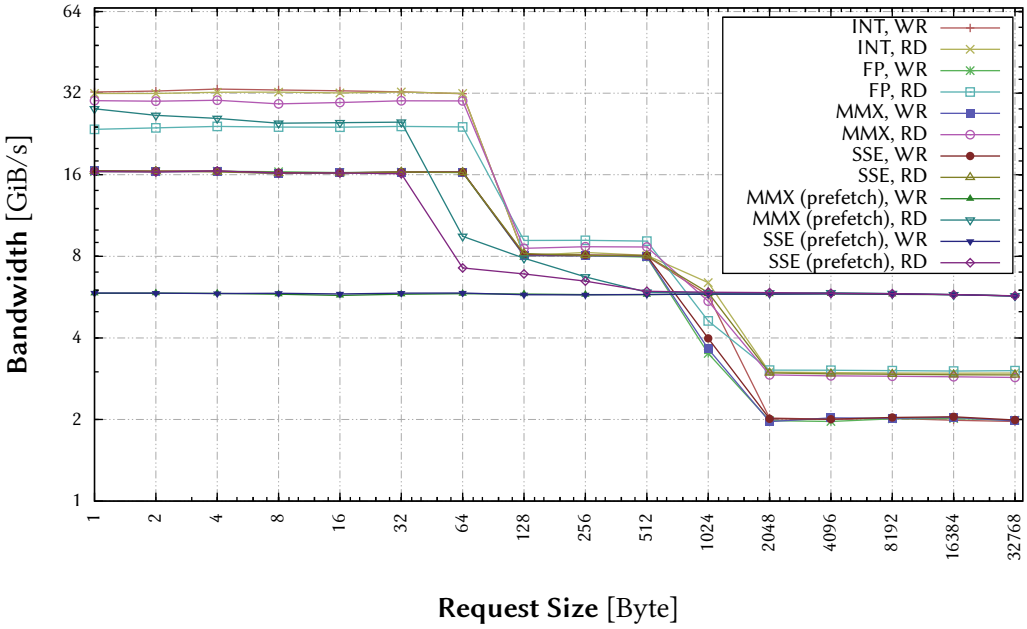


Figure 5.3: Results of the RAMspeed benchmark for different block sizes

16 bit = 1.6 GB/s ≈ 1.49 GiB/s. Each packet transmitted on the HT link consists of a header with a minimum size of 8byte and a maximum payload of 64byte, so a maximum of $\frac{64\,\text{byte}}{(64+8)\,\text{byte}}$ of the peak bandwidth is available for payload, which is approximately 1.325 GiB/s.

Figure 5.4 presents the bandwidth achieved when the CPU sends data to and reads data from the FPGA, respectively. The write bandwidth starts at about 0.25 GiB/s when sending 4 byte large blocks of data. It increases about linearly up to about 1.3 GiB/s for 24 byte large blocks, which is quite close to the theoretical peak value as depicted above. For larger transfer sizes, the bandwidth remains about constantly 1.3 GiB/s.

In contrast, the read bandwidth is very low for all packet sizes. Monitoring the HT link using JTAG resulted in that the host split the large data transfers into HT packets requesting 64 bit of data. The requests were not chained, each request had to be responded to before the next request was sent to the FPGA. This results in a very low utilization of the HT link, making this communication method inappropriate for large amounts of data. It is only suitable for reading single values from the FPGAs, for example the state of single registers.

5.2.3 Burst Read/Write Transfers Initiated by the FPGA

As pictured in Chapter 4 the FPGA in the XD1000 is able to directly access its own area of external DDR SDRAM as well as host memory. Host memory access is performed

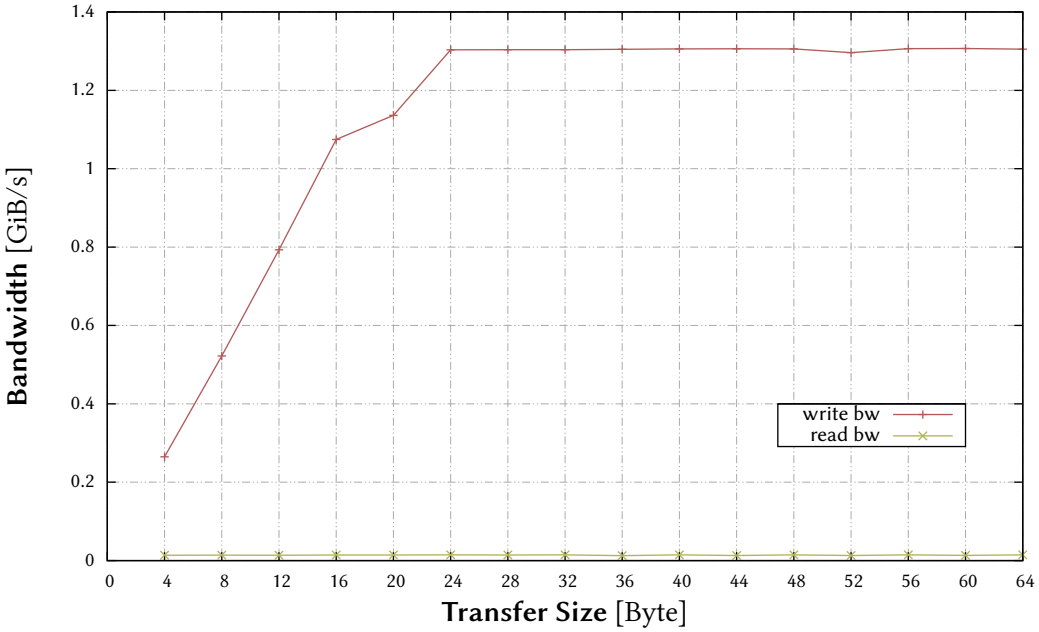


Figure 5.4: CPU↔FPGA communication bandwidth, communication initiated by the CPU

by sending appropriate HT packets to the corresponding links, which will likely provide the best performance if exploiting the maximum packet size possible. Additionally HT packets have to respect several alignment rules which may require transfers to be split into multiple packets even if their size matches the maximum HT packet size. DDR SDRAM access is done in bursts and will perform best when executing full burst transfers with the base address aligned to the burst size.

The first benchmark measures the peak bandwidth achievable when conducting appropriately aligned transfers with different packet sizes. For this it connects a benchmarking core with a datapath as wide as the memory's native width to the appropriate memory, e. g. 256 bit for the DDR SDRAM and 64 bit for the host memory. The benchmarking cores are running at 200 MHz, which is the host memory's maximum clock rate and more than the DDR SDRAM's clock. This configuration deployment ensures a full utilization of the memories. The benchmarking core was configured for generating read and write transfers with different IMORC packet sizes, respectively.

Figure 5.5 presents the resulting read bandwidth achieved on the host memory. The bandwidth nearly linearly increases with the packet size and reaches a maximum value of about 1.3 GiB/s at a size of 64 byte per packet. The rate then slightly drops to about 1.2 GiB/s and increases again to the maximum bandwidth at a packet size of 128 byte per packet. Considering the discussion of the HT packet size in the previous section, one draws

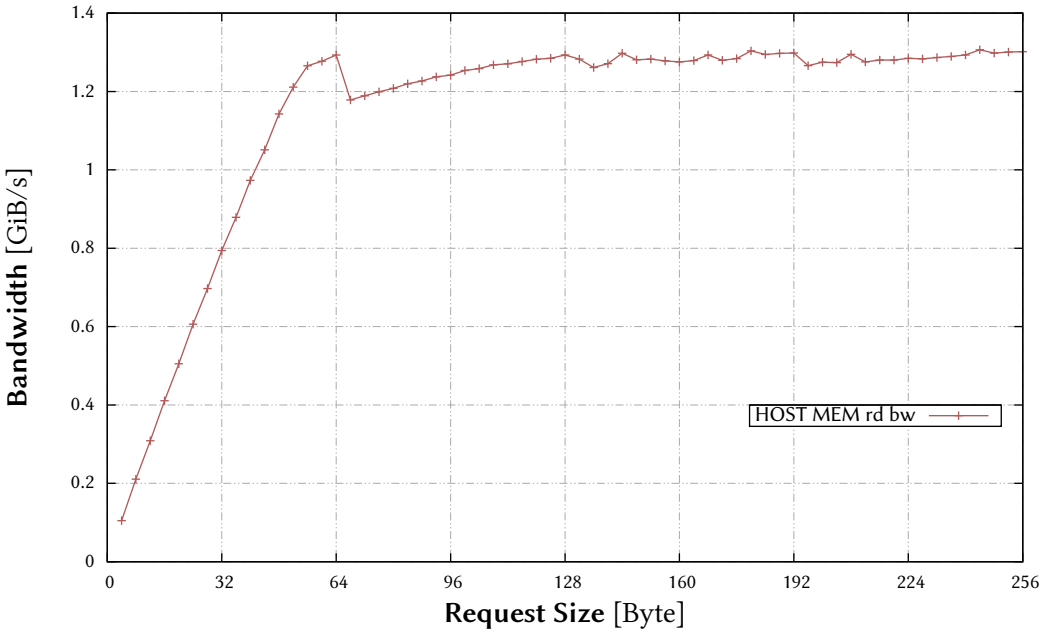


Figure 5.5: Read bandwidth achieved on the host memory, one core, 64 bit

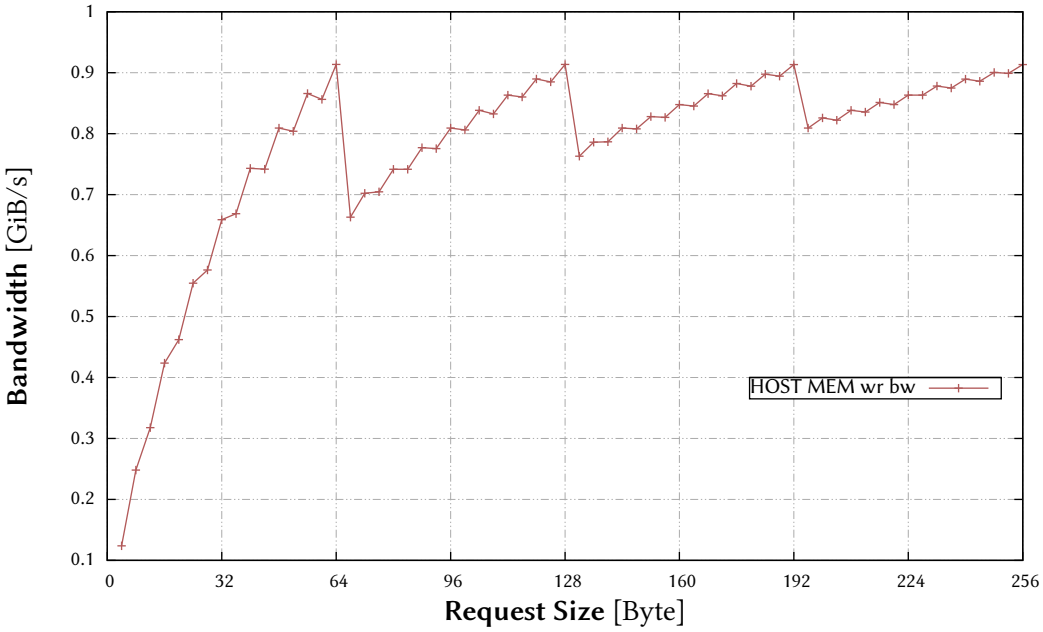


Figure 5.6: Write bandwidth achieved on the host memory, one core, 64 bit

the conclusion that the maximum bandwidth is achieved if the transfer size is a decimal multiple of the maximum HT packet size. The bandwidth achieved in this case is very close to the theoretical maximum bandwidth as discussed in the previous section.

A similar behavior can be observed when regarding the write bandwidth presented in Figure 5.6. However, the values achieved in this case are much lower than those achieved by the read benchmark — the maximum bandwidth with 64 byte large packets is at about 0.9 GiB/s and drops to about 0.7 GiB/s for packets slightly larger than this.

Figure 5.7 presents the read bandwidth achieved on the DDR SDRAM for different memory controller configurations with 2/4/8-cycle bursts, respectively. As expected, all cores perform best when the IMORC packet size matches the native burst size of the memory controller. The 4-cycle burst controller and the 8-cycle burst controller achieve about the same maximum performance of about 4.8 GiB/s. However, the 4-cycle burst controller already reaches this performance at its native burst size of 64 byte, whereas the 8-cycle controller does not get this performance before a request size of 128 byte. An interesting fact is that the graph of the 8-cycle controller matches the one of the 2-cycle controller for request sizes between 32 byte and 96 byte. Obviously, the controller stops the bursts and, thus, only performs 2-cycle bursts for request sizes less than 96 byte.

While the read benchmarks suggest the use of the 4-cycle controller in all cases, it is different for the write benchmarks presented in Figure 5.8. A heavy overhead is introduced due to the read-modify-write cycle inserted into the SDRAM controller. As a consequence, the performance is very low for non-aligned or non-full burst writes. Only when executing full-burst writes, the performance increases to about the same values as in the read benchmark. In order to achieve a good performance of the XD1000 processing storage tasks on its DDR SDRAM, aligned and complete burst writes are essential. In some cases, for getting a resonably high write performance, it may be necessary to only use the 2-cycle burst controller, even if its maximum read and write performance is only about half of the other two controller configurations.

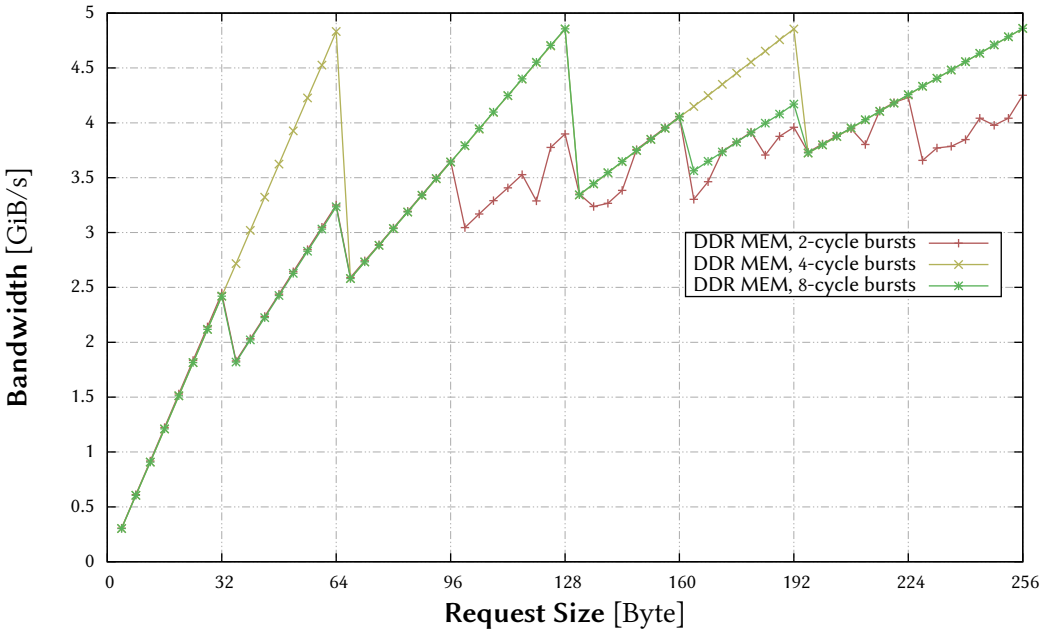


Figure 5.7: Read bandwidth achieved on the DDR SDRAM, one core, 256 bit

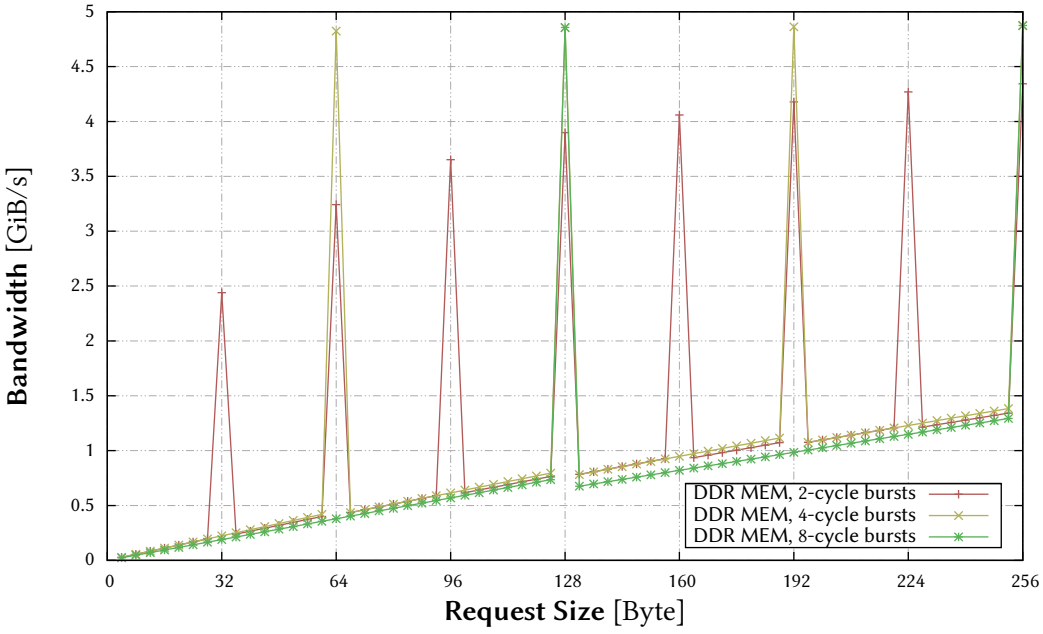


Figure 5.8: Write bandwidth achieved on the DDR SDRAM, one core, 256 bit

5.2.4 Simultaneous Access by Multiple Cores with a Common Access Scheme (Read or Write)

While the benchmarks presented above show the peak values achievable when accessing memories, the communication schemes found in real applications often do not match the communication scheme used in those benchmarks. It has to be especially taken into account that the benchmarks presented above were performed with a datapath's width matching the bitwidth of the memory. However, real-world cores often operate on data with a different bitwidth and accordingly may introduce some inaccuracy. Additionally, memory will likely be accessed by multiple cores in a real accelerator whereas the benchmarks presented above were performed with only one core accessing the memory.

For gathering characterization that more resembles real-world accelerators, further benchmarks were performed with multiple cores accessing the memory in parallel. Figure 5.9 presents the results of such a benchmark with four cores accessing the off-chip DDR SDRAM. The datapath of all four cores is 64 bit wide; the cores are running at 200 MHz. For simplifying the comparison with the results of the previous benchmarks, the values presented in the figure are the aggregate bandwidth available for all cores. The results strongly resemble the results of the benchmarks presented in the previous section, which draws the conclusion that the slave arbitration hardly introduces any overheads in this scenario.

Figure 5.10 presents the corresponding results for the write benchmark. In this case, all four cores concurrently write to the DDR SDRAM. The values presented again are aggregate bandwidth values of all four cores. the results also strongly resemble those of the previous benchmarks, virtually no overhead is introduced by the slave side arbitration.

Figure 5.11 shows a similar read benchmark using four cores accessing the DDR SDRAM, but this time the datapath of the cores is configured to a width of 32 bit. Since the theoretical aggregate bandwidth of the four cores (4×4 byte $\times$ 200 MHz = 2.98 MiB/s) this time is much lower than the measured peak bandwidth of the DDR SDRAM, the graph does not resemble a sawtooth curve. Instead, the bandwidth now approximately linearly increases up to about 2.7 GiB/s and then converges towards the maximum bandwidth achievable on slave side of about 2.98 GiB/s. All curves show a small drop of the bandwidth at a request size of 68 byte. The 2-burst and the 8-burst controller show an additional drop at a request size of 36 byte. Again, the measurements prove that the slave arbiters are working nearly optimally in this case introducing only little overhead. Figure 5.12 shows the corresponding graph of the write bandwidth. Due to the dramatically reduced bandwidth for non-full burst transfers, the graph greatly resembles the one using the 64 bit datapath, with the maxims cut to about the maximum theoretical bandwidth achievable on the slave side.

Other benchmarks performed have shown that the bandwidth achieved can further be increased by adding more benchmarking cores to the setup. Using seven cores, the

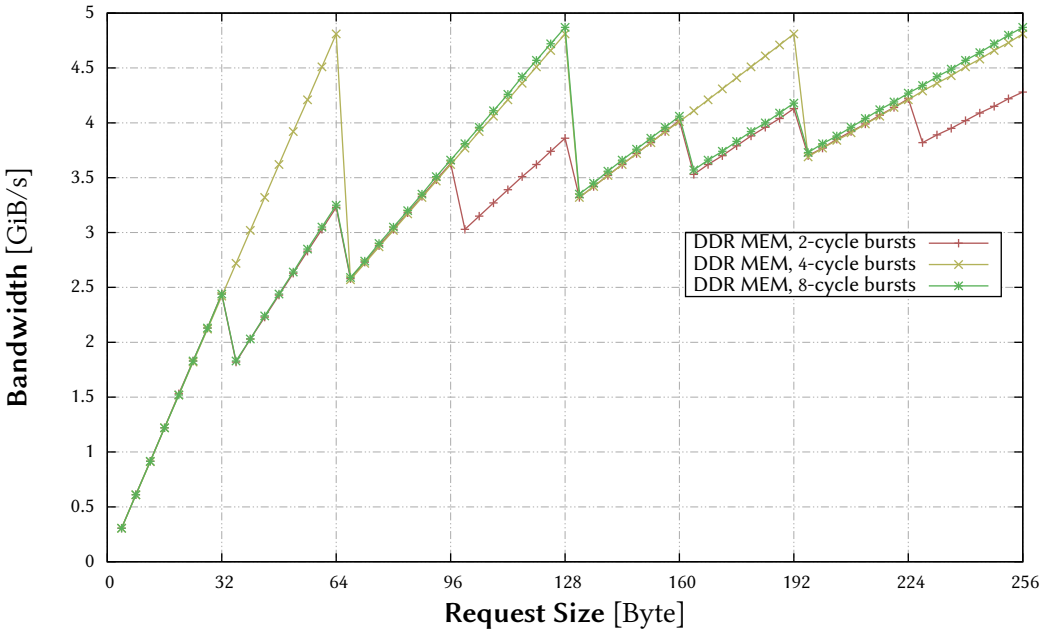


Figure 5.9: Read bandwidth achieved on the DDR SDRAM, four cores, 64 bit

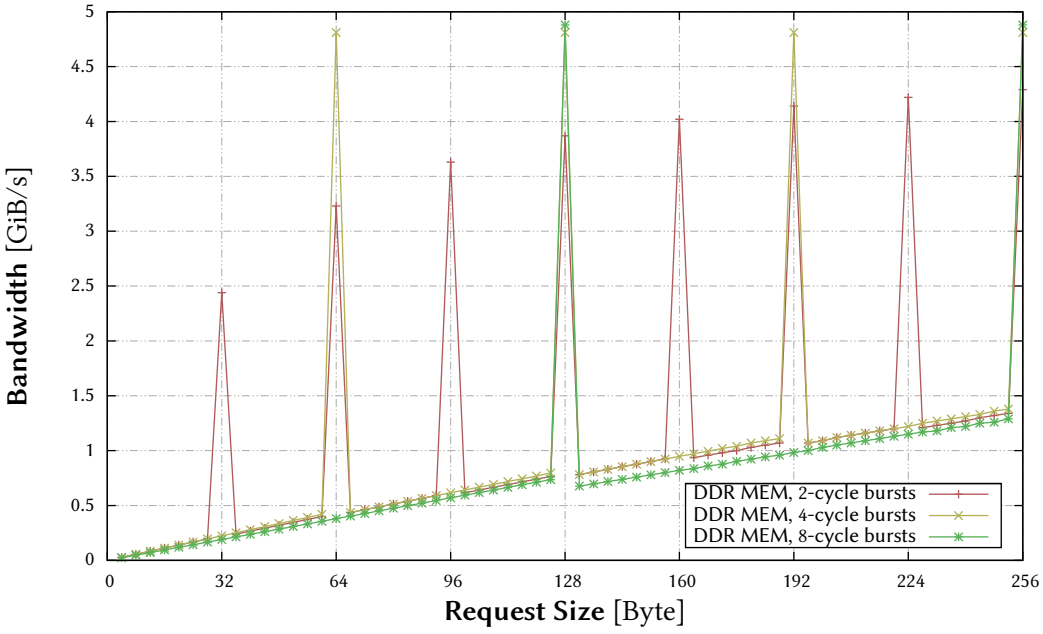


Figure 5.10: Write bandwidth achieved on the DDR SDRAM, four cores, 64 bit

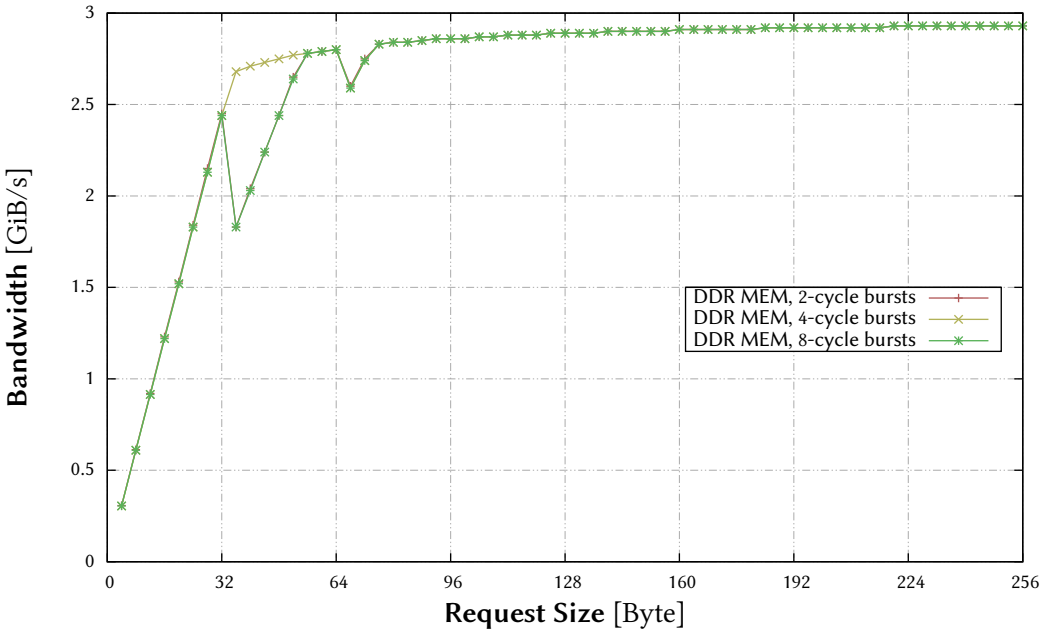


Figure 5.11: Read bandwidth achieved on the DDR SDRAM, four cores, 32 bit

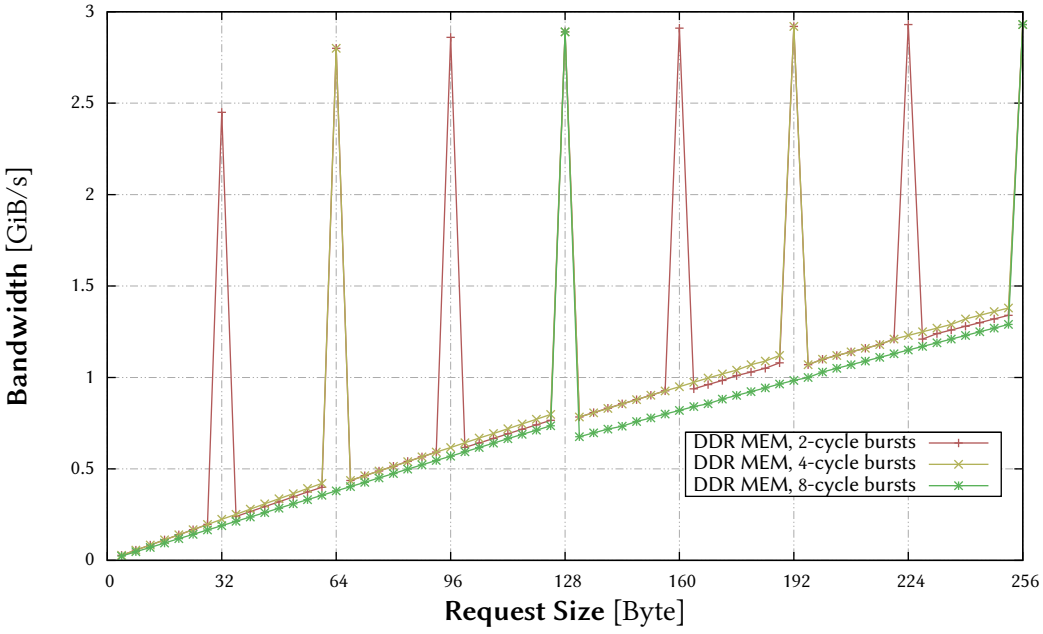


Figure 5.12: Write bandwidth achieved on the DDR SDRAM, four cores, 32 bit

benchmark achieves about the peak performance of the DDR SDRAM, driving it into saturation.

5.2.5 Contention Benchmark with Multiple Simultaneous Reads and Writes

The benchmarks presented above were configured so that all cores execute memory accesses using the same access scheme. Precisely, all cores were configured to perform reads or writes with the same request sizes. While this provides a detailed insight into the performance achievable in certain situations, real world cores will likely not only read or write data but first read some input data, process it and write the results back. The DDR SDRAM uses the same bus for reading and writing data from/to the memory. Consequently, the combined read/write bandwidth cannot be higher than the maximum unidirectional bandwidth. The HyperTransport used as host interface contrariwise provides two distinct unidirectional links between the host CPU and the FPGA. As a result, it should theoretically be possible to achieve the maximum read bandwidth as measured in the previous benchmarks and to write to the host memory at the maximum write bandwidth measured at the same time.

For proving these assumptions, additional benchmarks were performed with a similar setup as presented in the previous section. The main difference between these benchmarks is that now half of the cores is configured to perform memory reads, the other half is configured to perform memory writes.

Figure 5.13 shows the resulting bandwidth values for the DDR SDRAM with the four different controller configurations. Four cores with a 256 bit wide datapath were accessing the memories, two of them performing read operations, the other two performing write operations. The graph resembles the one of the write benchmarks presented above. However, the underlying curve for non-aligned request sizes ascends more steeply than in the write-only benchmarks. Additionally, the peak values achieved when performing burst size aligned transfers is decreased to about 4.4 GiB/s for the 4- and 8-cycle burst controllers and to about 4.2 GiB/s for the 2-cycle burst controller. Contrary to the previous benchmarks, this peak performance is not already achieved at the first request size matching a full burst. Instead, the rate for full burst transfers increases continuously with the request size.

The corresponding benchmark for the host memory is presented in Figure 5.14. Analogous to the other results, the bandwidth measured increases with the request size to nearly 1 GiB/s for requests slightly smaller than the HT's maximum packet size and then jumps up to about 1.65 GiB/s for requests matching a complete HT packet. It then decreases again to about 0.6 GiB/s, ascends to about 1 GiB/s and jumps up again to about 1.3 GiB/s for requests matching two complete HT packets. This value is also the peak bandwidth achieved for higher request sizes matching an integer multiple of complete HT packets. The reason for this behavior is currently not completely clear. Monitoring the user interface

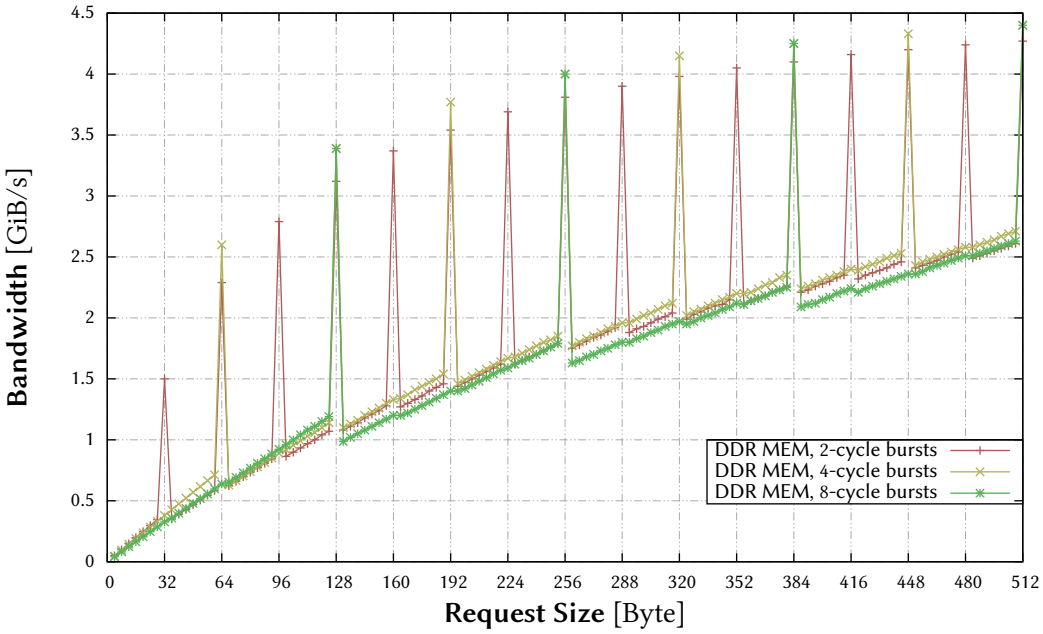


Figure 5.13: R/W bandwidth achieved on the DDR SDRAM, four cores, 256 bit

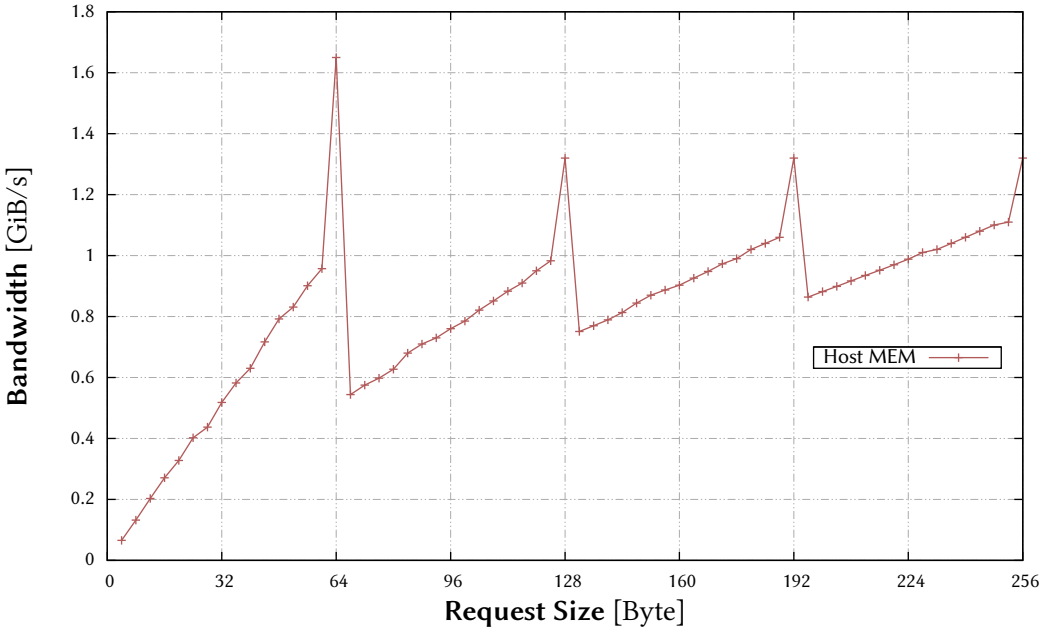


Figure 5.14: R/W bandwidth achieved on the host memory, four cores, 64 bit

of the HT cave using the IMORC load sensors shows that on the one hand packets are injected into the HT cave as soon as the cave is able to accept requests. On the other hand data is read from the response queue as soon as it becomes available. Hence IMORC completely utilizes the bandwidth provided by the HT cave.

5.3 Chapter Summary

This chapter introduced the IMORC benchmarking infrastructure and presented a detailed characterization of the different communication channels the XtremeData XD1000 system provides. While naturally the performance achieved on each of these communication channels is best when the communication pattern is adopted to the concrete communication channel, such adoptions are not always possible in real world accelerators. Consequently, a detailed characterization is necessary for analyzing the performance achievable by such accelerators. The sparse matrix multiplication kernel presented as an example in section 3.3.2 transforms all memory accesses into a stream of data to be processed. However, in an extreme case the vector to be multiplied with has to be accessed in portions of 32 bit or 64 bit, depending on the floating point format used. Such access will perform very poorly on both the host memory and the DDR SDRAM, which makes an accelerator only suitable when the vector is accessed blockwise.

While the benchmarks already provide a good insight into the communication performance of the XD1000, estimating the achievable speedup for concrete applicatons may require different sets of benchmarks. The benchmarking infrastructure is in that case flexible and configurable. Hence other kinds of workloads can be easily simulated.

The benchmarks also point out the main design goal of the IMORC architectural template: The infrastructure provides a very high performance with only little overhead, with the result that many cores accessing the same memory achieve about the same peak performance physically achievable on the concrete target memory.

6 Experimental Evaluation

This chapter deals with the design and implementation of some real world accelerators using the IMORC workflow. The design and optimization of the accelerators are based on the results presented in Chapter 5. The case studies demonstrate the suitability of the IMORC development flow. Each case study starts with a problem description, followed by an analysis of the algorithm and the design of the accelerator using the IMORC modeling flow. The implementations based on the IMORC architectural template express the usefulness of the infrastructure and of the supporting cores for reconfigurable accelerator development.

Three case studies taken from distinct problem domains are being evaluated: The first one demonstrates an accelerator for a kernel of the Cube Cut problem. The kernel performs a high number of independent bitwise operations on a large amount of data, which is intuitively a problem class that should perform well on FPGAs. The second case study focuses on an accelerator for a compositing kernel used in a parallel rendering framework. In contrast to the first case study, the algorithm is not qualified for being implemented by using a long computation pipeline. Therefore, the performance compared to that of a commodity CPU greatly depends on the time needed for memory access. The third case study presents an accelerator for the k-th nearest neighbor thinning problem. The accelerator consists of several communicating cores, showing the use of the supporting cores as well as the use of infrastructure cores like, for example, the farming cores. The accelerators are optimized by using the IMORC performance counters, which especially give an in-depth insight into the runtime behavior of the complete accelerator in the third case study.

6.1 Cube Cut

A d-dimensional hypercube consists of 2^d nodes connected by $d \times 2^{d-1}$ edges. A familiar and still unsolved problem in geometry is to determine $C(d)$, the minimal number of hyperplanes that are required to slice all the edges of the d-dimensional hypercube. Intuitively, an upper bound for $C(d)$ is given by d. Such a cut can, for example, be generated by spanning d hyperplanes which are normal to the unit vectors through the origin of the hypercube. While for $d \leq 5$ it is known that at minimum d hyperplanes are required, it was shown that for $d = 6$ actually only five hyperplanes are required [114].

The cube cut problem is related to the question of linear separability of vertex sets in a d-hypercube, which plays a central role in several different areas such as threshold logic [63], integer linear programming [36] and perceptron learning [93]. In the past, much effort was put into finding $C(d)$ [53, 94, 103]. Basically two different approaches are followed: an analytical solution and a computational solution. In the analytical solution scientists try to find a mathematical proof for $C(d)$. Such a proof would be desirable, but seems to be hard to obtain even for constant values of d. This led to the development of a computational evaluation of the problem — given a d-dimensional hypercube, the computational solution first generates all possible slices of the hypercube and then uses exhaustive search methods to find the minimal subset of these slices cutting all edges of the hypercube. This method lacks the mathematical strength of a formal proof, but has indeed achieved a remarkable success during the past years [114, 124].

The large number of possible slices for higher dimensions however produces very long runtimes to identify the minimal subset of slices cutting all edges of the d-hypercube. To speedup the process, an additional step is introduced for reducing the data volume that has to be searched. This case study presents an accelerator for the data reduction part of the cube cut algorithm. The accelerator's design is based on previous work which is presented in [2] that implemented such an accelerator for a cluster equipped with four AlphaData ADM-XP FPGA boards connected to the host system using PCI 64/66. The accelerator presented in this section is different in that way that it is based on the IMORC architecture template and that it provides more flexibility regarding the throughput achieved on different platforms.

6.1.1 The Cube Cut Algorithm

The computational solution to the cube cut problem can basically be separated into three distinct phases:

1. For the d-dimensional hypercube, all possible cuts are generated. The cuts can be represented by the edges it slices, e. g., for the d-dimensional hypercube containing $n = d \cdot 2^{d-1}$ edges a cut can be described by a string of n bits. Each bit position in that

string corresponds to a specific edge and is set to '0' if the edge was not sliced by a cut and to '1' if it was sliced. For example, for $d = 2$ a cut is described by a string of 4 bits. For greater values of d, the number of bit strings found can become extremely large. However, there are principally two ways to reduce the amount of data to be stored: first, symmetrical cuts are only stored once, e. g., for $d = 2$ ($\{1010\}, \{0101\}$) would be reduced to $\{1010\}$ since the second cut can be generated from the first one by shifting the dimensions. This reduction in storage size can easily be executed during generation of the cuts.

2. When the first phase of the algorithm is finished, the data is further reduced by finding cuts dominating others. Consider a cut a that slices the edges $E = \{e_0, e_1, \ldots, e_{k-1}\}$ and another cut b that slices all edges in E plus some additional ones. If a is selected as a candidate for this minimal subset it can be simply replaced by b since the major goal is to find a minimum subset of cuts slicing all edges of the d-hypercube. b is said to *dominate a*. Formally:

$$b \text{ dominates } a \iff \forall i : (\neg a_i \vee b_i) = 1; i = 0, ..., n - 1$$

In the second phase the list of cuts is therefore searched for all cuts that are dominated by another cut. The cuts dominated are removed from the original list of cuts. This is realized by first generating two separate lists A and B from the initial list of possible cuts. B is initialized with all bit strings that slice a maximal number of edges — these bit strings can consequently not be dominated. Assuming that every element of B contains exactly k_{max} ones, list A is initialized with all bit strings containing $k_{max} - 1$ ones. Every element in A is compared to every element in B; the elements of A that are dominated are discarded. Next, the elements of A that were not dominated are added to list B. List A is now initialized with all bit strings containing exactly $k_{max} - 2$ ones. This procedure is repeated until all elements of the original list have been checked for dominance. The result of this algorithm is a reduced list of bit strings whose irrelevant cuts all have been discarded.

3. The last phase of the algorithm is to search the reduced list of bit strings for a minimal subset of cuts that slice all edges of the d-hypercube. The bit strings omitted in the first phase due to symmetries to other cuts are considered again — they can easily be generated from the reduced list of bit strings by shifting and/or reversing the bit strings available in the list. Taking the example of hypercube with $d = 2$, bit string $\{1010\}$ can be shifted right by one dimension to reconstruct the second bit string. The bit strings are combined using a bitwise `OR` operation. This results in a bit string with all bits set to '1' — in other words, a set of two cuts was found where each edge of the hypercube had been sliced at minimum once. Since the list does not contain a single cut with all bits set to '1', this set of two cuts forms a minimal set of cuts slicing all edges of the hypercube.

Regarding the outlined algorithm, one might argue that removing bit strings due to symmetrical properties in the first phase and adding them again in the last phase introduces a certain overhead. However, it is on the one hand rather simple to omit these bit strings in the first phase (i. e. to not generate such bit strings at all) and to generate them in the last phase. On the other hand, it significantly reduces the amount of storage needed for the list of cuts and therefore also extremely decreases the runtime of phase two of the algorithm.

Although phase two of the cube cut algorithm consists of rather uncomplicated bit operations on bit strings, it needs a very high runtime in software. This is mainly due to the fact that the required bit operations and the lengths of the bit strings do not match the instructions and operand widths of commodity CPUs. FPGAs however can be configured for operating directly on a complete bit string at once and hence exploit the high potential of fine-grained parallelism. Moreover, if sufficient hardware area is available, many of these operations can be executed at the same time with rather small overhead. An FPGA implementation can therefore also leverage both the fine- and coarse-grained parallelisms inherent in phase two of the cube cut algorithm.

6.1.2 Design and Implementation

Algorithm 6.1 outlines the basic dominance check algorithm to be implemented. In a first step of the design process, a dominance check task is defined. According to the nomenclature in Section 6.1.1 the task's input parameters are a b-value and a stream of the bit strings in A. Each element in A is compared to the b-value and written back if not discarded. Then, the task is configured with another b-value and again each element in the reduced list A is compared to this value. This procedure is repeated for all $b \in B$.

1: **for all** $b \in B$ **do**
2: **for all** $a \in A$ **do**
3: **if** $\bigwedge_{i=0}^{n-1} (\neg a_i \vee b_i) = 1$ **then**
4: $A \longleftarrow A \setminus a$
5: **end if**
6: **end for**
7: **end for**

Algorithm 6.1: Dominance check of the cube cut algorithm

To exploit parallelism in the next step multiple dominance check tasks are chained. The taskgraph displayed in Figure 6.1 visualizes this chaining. First, every task is configured with a value $b_i \in B$. Then, the bit strings in A are streamed through the pipeline and every task compares each value in A to b_i. The maximum pipeline depth depends on the

resources available in the FPGA. The number of iterations to be performed depend on the pipeline depth and the number of bit strings in B.

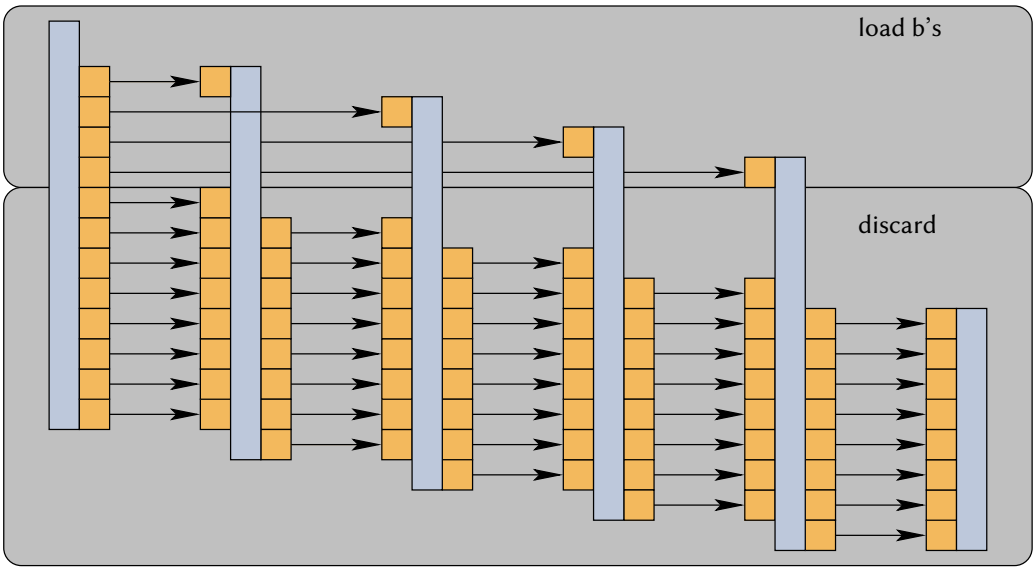


Figure 6.1: Task graph of the cube cut algorithm

With a clock frequency of c and 192 bit = 24 byte wide data this design would create a bandwidth demand of $\beta = c \cdot 24$ byte on the input side of the pipeline and a similar demand on the output side. This would exceed the bandwidth available on the XD1000 even for low clock frequencies.

To increase the utilization of the pipeline and to reduce the number of iterations to be performed for a given input data set, the dominance check task is converted into a multicycle task. This contains some local memory for storing multiple values of B — each value from set A is now compared to the complete subset of B available in this local storage.

The maximum depth of the pipeline depends on the number of logic resources and the amount of embedded memory blocks available in the FPGA. The number of cycles each a stays in one stage of the pipeline depends on the local memory depth, which depends on the type of embedded memory blocks the local storage is mapped to.

With a clock frequency of c, 192 bit = 24 byte wide data and a pipeline latency of l, the incoming bandwidth demand and upper bound for the outgoing bandwidth demand of the pipeline can be evaluated to

$$\beta = \frac{c \cdot 24\,\text{byte}}{l}\,.$$

In this design, the number of memory resources available will likely be the restricting parameter for the pipeline depth because the dominance check itself is a rather simple

operation that can be implemented with a small number of logic resources. To increase the amount of logic resources occupied again and to introduce a further parameter for optimizing the throughput of the pipeline, multiple pipelines can be implemented in parallel, sharing the same local memory. Using m parallel pipelines, m a-values are compared to one b in parallel, increasing the maximum throughput of the pipeline by factor m. This results in an input bandwidth and an upper bound for the output bandwidth of

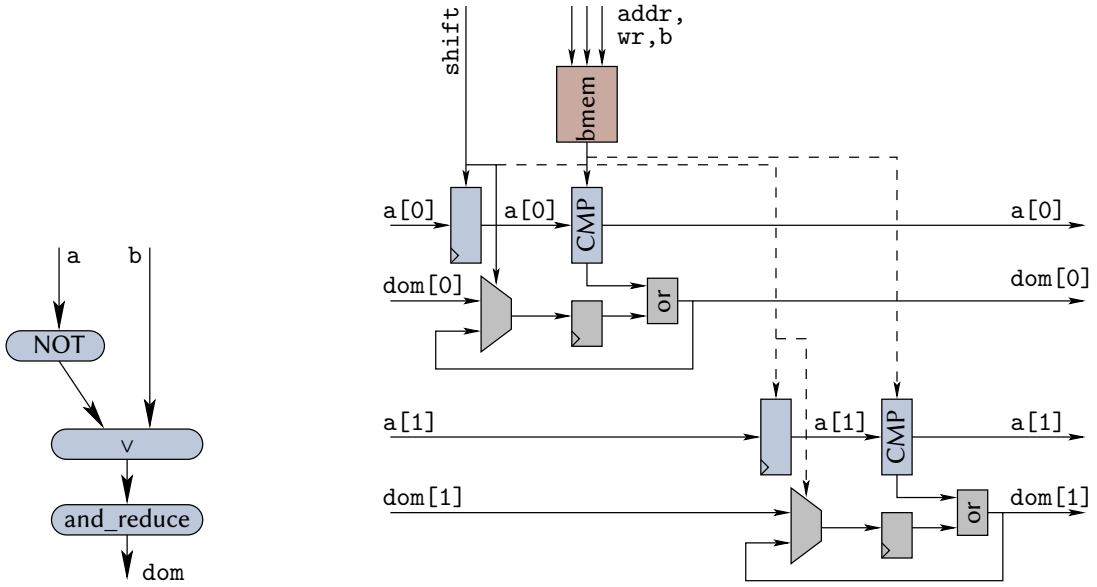
$$\beta = m \cdot \frac{c \cdot 24\,\text{byte}}{l}\,.$$

6.1.3 Architecture Mapping, Implementation and Performance Evaluation

The main compute core of the cube cut accelerator implements the dominance check task. While the original design presented in [2] explicitly instantiated the slice's carry logic on the Xilinx Virtex II Pro FPGA, the new version is completely implemented using vendor neutral VHDL code. The comparator takes as input two 192 bit wide vectors a and b, and calculates a third vector by bitwise performing the operation $\delta = \neg a \vee b$. An `AND` reduction operator is applied to vector δ. If the result of this reduction evaluates to '1', vector b is said to dominate vector a.

Figure 6.2 shows the dataflow graph of the dominance check operator (a) and one stage of the cube cut pipeline (b) with two comparators connected to one local memory block. Each dominance check task of the task graph presented in the previous section is mapped to one instance of the dominance check pipeline stage. The core provides an `addr`, `wr` and `b` signal for writing bs to the internal memory block. The `addr` signal is also used for selecting the b to be compared to the current a stored in the register. Signal `shift` is used for shifting as from one stage of the pipeline to the next stage. The signal is used as an enable signal for the register storing the a and as a select signal for the input to the dominance register. The dominance register stores whether the a was already dominated by one of the bs it was compared to previously, thus marking the a as invalid.

For mapping the local memories of the pipeline stages, three different types of embedded memory blocks are available as listed in Table 6.1. Additionally, the table lists those parameters that are important for mapping the local storage tasks to the different kinds of memory. The M-RAM blocks are the largest ones, but are not adequate for the desired purpose due to the low number of available resources. M512 blocks would allow to implement up to 77 pipeline stages with a latency of 32 clock cycles per pipeline stage. M4K allows 128 pipeline stages with a latency of 128 cycles. These values are upper bounds, since additional memory blocks are needed for IMORC's embedded FIFOs and the host communication interface. To reduce resource overheads, lists A and B will be stored in host memory; consequently, no DDR memory controller is required. The local memories



(a) Dataflow graph of the dominance check

(b) One stage of the dominance check pipeline

Figure 6.2: Block diagrams of the dominance check element

are implemented in vendor neutral VHDL and configured to store 128 values. This way, portability is ensured while on the current target platform the synthesis tool will map the memories to M4K blocks.

	M512	M4K	M-RAM
# blocks	930	768	9
widest config	32 × 18	128 × 36	4 K × 144
blocks/192 bit	12	6	2
max pipeline stages	77	128	4

Table 6.1: Different types of memories available in the Stratix II EP2S180 and their parameters

The maximum number of parallel pipelines and the maximum clock rate achievable using this architecture were evaluated to $m = 5$ and $c = 100$ MHz in several synthesis runs. With $m = 5$ parallel pipelines, the number of registers occupied by the complete pipeline grows up to 127150, which is 89 % of the register resources available in the FPGA. With these parameters and a pipeline depth of 128 stages, the input bandwidth demanded

evaluates to

$$\beta_{in} = 5 \cdot \frac{100\,\text{MHz} \cdot 24\,\text{byte}}{128} = 89.4\,\text{MiB/s}\,,$$

which is much less than the maximum read bandwidth achievable on the host memory. Assuming that in the worst case no bit strings are removed in an iteration, i. e., the output bandwidth of the pipeline is equal to the input bandwidth, the achievable aggregate reading and writing performance on the host memory is much higher than the aggregate input and output bandwidth demand generated by the pipeline. Hence, the original, intermediate and final problem data can completely reside in the host memory, avoiding the need of an additional DDR memory controller.

The control core wraps the control task of Figure 6.1 and is responsible for loading *b*s into the pipeline's memory blocks, for sending *a*s into the pipeline and for writing the *a*s, which are not discarded, back to the host memory. It consists of an I2R interface core decoding job parameters, a request generator core for sending requests to read *a*s and *b*s from host memory and another request generator core for sending write requests to host memory in order to store non-dominated *a*s back. Custom logic is added for controlling the current state of the accelerator. The accelerator is started with a job request sent to the I2R interface core, consisting of the base address of lists A and B in host memory and the number of vectors available in each list. The read request generator core is instructed to send read requests to gather the appropriate amount of values from list B. Values received are written to the corresponding locations in the pipline's embedded memory blocks. When finished, the read request generator is instructed to fetch all values from list A which are then forwarded to the pipeline. The write request generator is instructed to send write requests to the location of list A. Since the number of values to be written back is not known in advance, this request generator waits for values to be written to the M2S channel and sends the corresponding requests as soon as a configurable amount of data is written back. When all *a*s are processed, the next bunch of list B is transferred from the host memory to the pipeline's memory blocks and all values that were written back to list A in the last iteration are again fed through the pipeline. This procedure is repeated until all values from list A are checked for dominance by all values from list B.

Figure 6.3 shows the diagram of the complete accelerator including the control block and the pipeline. Including the control core, the overall design takes up 65 859 ALUTs (46 %), 101 933 Registers (92.2 %), 721 M4K blocks (93.9 %) and 201 M512 blocks (21.6 %) — the values in brackets represent relative values to the overall resources available in the FPGA. The synthesis tool did not to place all local memories of the pipeline stages into M4K blocks, probably to ensure better routability.

To verify the design presented in the previous sections and to determine its performance, the accelerator was tested with real data generated by the cube cut algorithm. The dataset consisted of list A with 5 339 385 strings containing 147 ones and list B with 409 685 strings containing 148–160 ones. Running on the CPU of the XD1000 system, the software

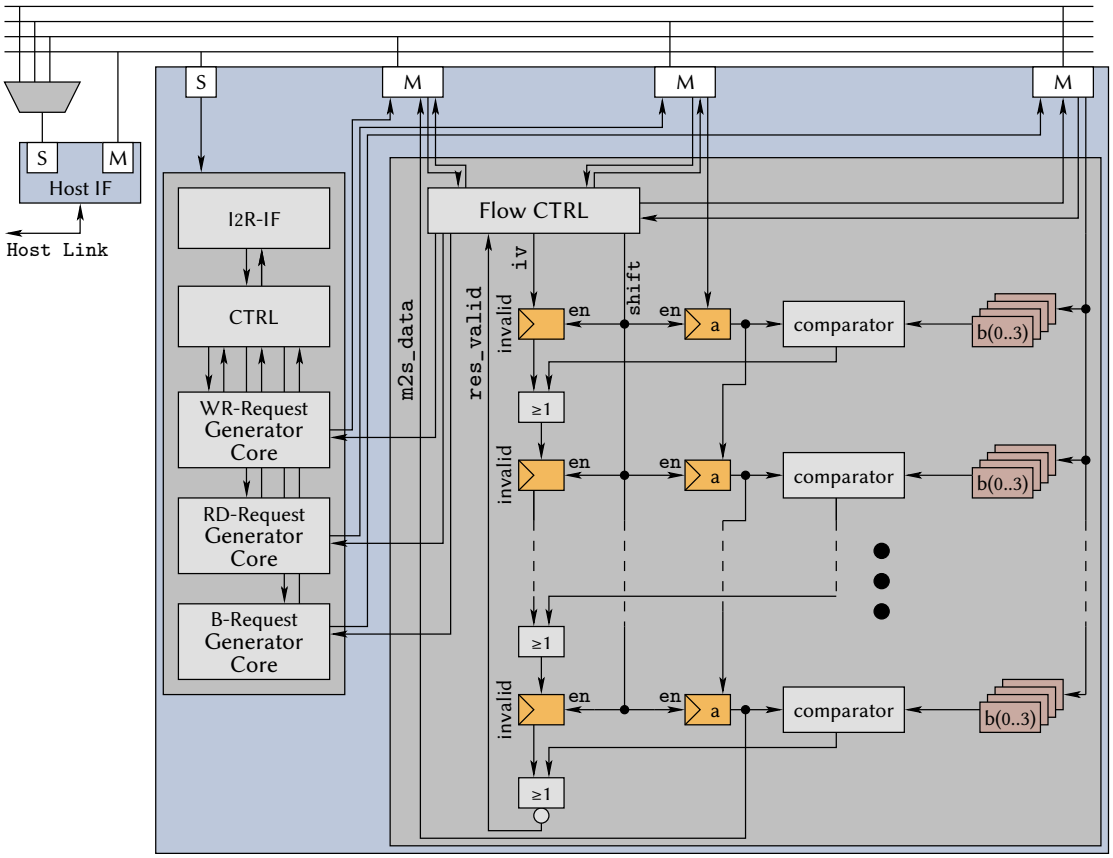


Figure 6.3: Architecture diagram of the complete cube cut accelerator

solution finished after 3281 seconds while the runtime of the accelerator only took 9 seconds, resulting in a speedup of factor 365.

6.2 A Compositing Accelerator for a Parallel Rendering Framework

The second case study presented implements a compositing accelerator for a parallel rendering framework [7, 8]. 3D Computer Aided Design (CAD) applications such as production planning and optimization and mechanical component design nowadays use highly detailed object models originating from CAD tools or 3D scanners. Engineers can move around in the scene, rotate and move objects and zoom in or out. To ensure a smooth visualization of such scenes with huge numbers of polygons, substantial computations have to be performed. Parallel rendering approaches were developed to fulfill the stringent

performance requirements. Molnar et al. [88] introduce three approaches for parallel rendering with different advantages and drawbacks: sort-first, sort-middle and sort-last.

This case study focuses on an in-house parallel renderer using the sort-last approach. A master node divides a scene (frame) into $N - 1$ subscenes and distributes the workload to $N - 1$ rendering nodes. The subscenes roughly have the same number of geometry primitives (polygons), but the assignment of polygons to rendering nodes is arbitrary. Each rendering node runs a rendering pipeline that computes a set of display primitives (pixels) from the received geometry primitives. A rendering node computes two buffers for its subscene, the frame buffer containing the color information and the z-buffer containing the depth information for each pixel. The resulting $2 \cdot (N - 1)$ buffers are transferred back to the master node for compositing. Compositing performs the sorting step by comparing the distances of the $N - 1$ candidates for each pixel to the view plane. Only the nearest display primitive is visible.

The parallel renderer is part of a visualization application that can be run in batch-mode or interactively. The master node stores or displays the composited picture and distributes the next subscenes to the renderer nodes. The application is implemented with MPI using double buffering for frames to overlap computation and network communication.

6.2.1 Application Model

The performance of the complete rendering application basically depends on the following parameters:

- H and W are the height and the width of a subscene (frame) in pixels. Each rendering node processes a frame buffer and a z-buffer for each subscene, resulting in $P = W \times H \times 8$ bytes of data.
- T_R [s/frame] is the time required for one rendering node to compute its subscene. T_R depends on the size of the subscene, i.e., on P and the number of polygons. Since the workload is evenly distributed, an average value for all rendering nodes can be used.
- T_C [s/frame] measures the computation time for the master node and comprises the compositing time and the time needed to display or store the resulting picture and redistribute the next workload. The compositing time is dominating and depends on P and the number of rendering nodes, $N - 1$.
- B_{net} [byte/s] is the bandwidth of the link over which the master node connects to the computer network.

The aggregated data bandwidth generated by the rendering nodes is $(N-1) \times P/T_R$ [byte/s]. Depending on P, the complexity of scenes, and the parameters of the compute cluster, a reasonably designed and configured system will try to set the number of rendering nodes

```
1  void compose(int* pic_a,int* pic_b,int size) {
2    int *z_a=pic_a+size;
3    int *z_b=pic_b+size;
4    for(int i=0;i<size;i++) {
5      if(z_b[i]<z_a[i]) {
6        pic_a[i]=pic_b[i];
7        z_a[i]=z_b[i];
8      }
9    }
10 }
```

Listing 6.1: Code for the compositing function

so that the network or the master node is not saturated. Bottlenecks occur if the aggregate renderers' bandwidth exceeds B_{net} or if the master node's computation time T_C limits the throughput. Benchmarks on different platforms have shown that the latter is currently more likely in practice which makes compositing an interesting target for acceleration.

Listing 6.1 shows the pseudocode for the compositing function. The function is computationally very simple, only comprising a regular loop with comparisons and assignments which can be parallelized in a straight-forward way. However, the main challenge for accelerating this code is to establish a continuous stream of data through the computing core. The compositing function outlined in Listing 6.1 basically needs up to five different storage locations:

- One location for each of the frame- and the z-buffers as received from the rendering nodes,
- one location for the intermediate frame- and z-buffer, and
- one location for the final frame buffer to be displayed.

The first two locations, the frame- and z-buffers received from the rendering nodes, are initially available in the host memory since the OpenMPI implementation [20] is unable to perform MPI receives directly into the FPGA's address space. The last location also needs to be available in host memory for being displayed. The intermediate frame- and z-buffer are exclusively accessed by the accelerator, hence can be mapped to FPGA-local memory resources. On-chip memory is not available in reasonable capacity for storing buffers in high resolution, so these buffers are mapped to the external DDR SDRAM.

Figure 6.4 shows a diagram of the memory access scheme of the compositing accelerator in three different phases. Overall, the accelerator has to transfer $(N-1) \times P$ bytes of data from host memory to the FPGA module. The first frame is directly transferred to the external memory. Since $B_{rd}(HM)$ is lower than $B_{rd}(EM)$, the time needed for this first phase

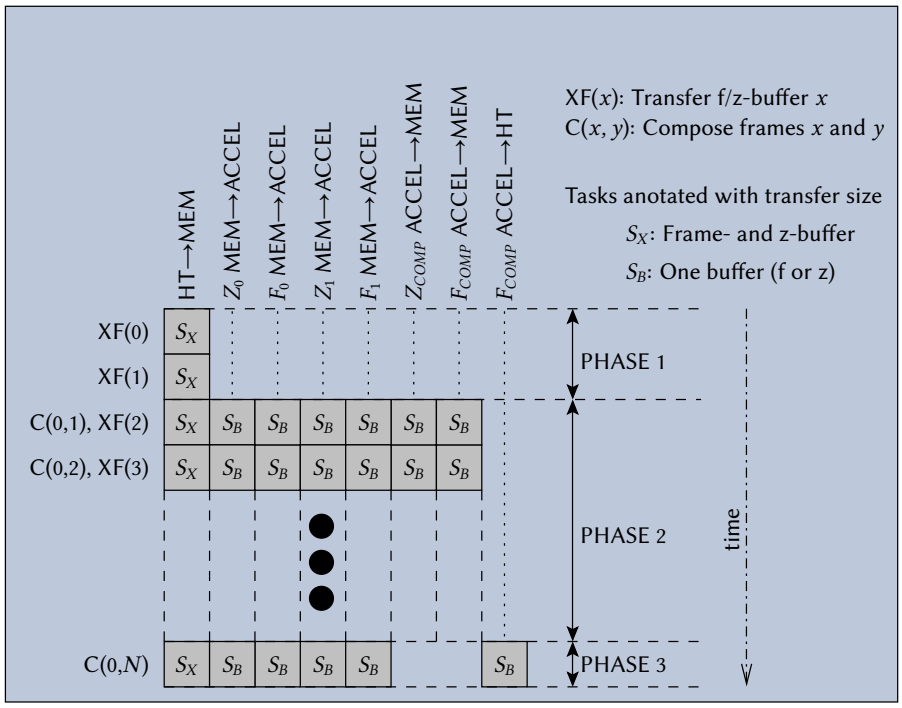


Figure 6.4: Memory access scheme of the compositing accelerator

of the accelerator is determined by the HyperTransport performance. The following $N - 2$ frames of size P stream from the host to the FPGA accelerator and, at the same time, the stored frame streams from external memory to the FPGA. The resulting frame streams back to external memory. The execution time required for this second phase is dominated by memory accesses and given as $\frac{P}{B_{rd}(EM)} + \frac{P}{B_{wr}(EM)}$ for the external memory and $\frac{P}{B_{rd}(HM)}$ for the host memory read over the HyperTransport link. Consulting the bandwidth measurements of Section 5.2, i.e., Figures 5.6 and 5.5, one draws the conclusion that for reasonably chosen request sizes the host memory access limits the execution time. This applies only if external memory is accessed with request sizes that are multiples of the controller's burst size. The actually chosen burst size of the memory controller influences the access time for external memory, but has no effect on the overall compositing application.

For the last frame, P bytes are read from each host memory and external memory but only $P/2$ bytes are written back to host memory since the resulting z-buffer is not needed for displaying the picture. Despite the fact that the write bandwidth to host memory is much lower than the read bandwidth, the execution time of this last accelerator phase is determined by reading the host memory since writing involves only half of the data size.

Figure 6.5 shows a comparison of the execution times for the FPGA compositing accelerator based on these estimations with the measured CPU execution times for the same task

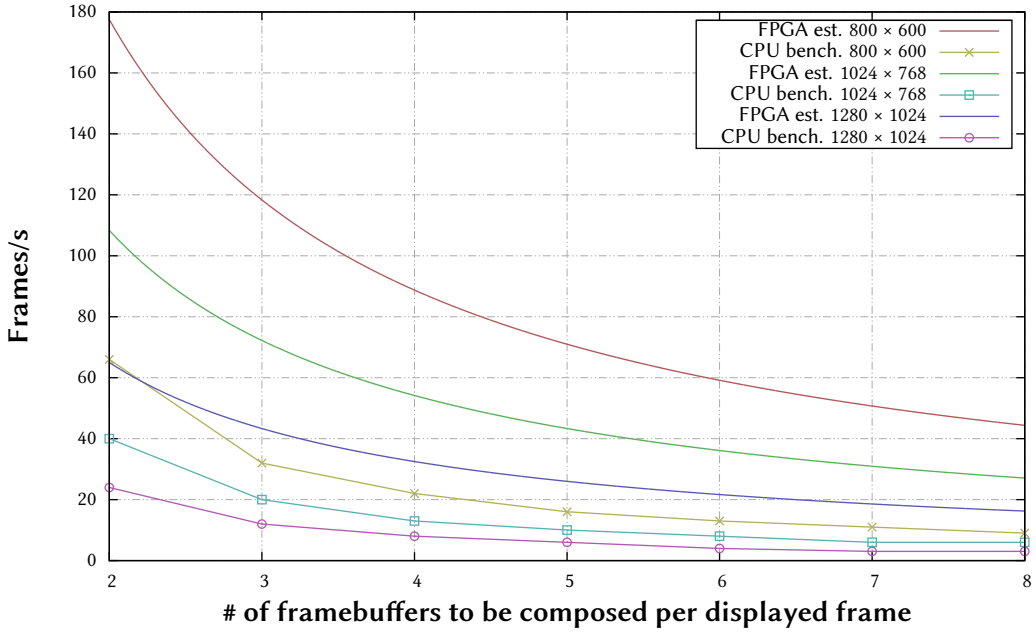


Figure 6.5: Comparison of CPU performance and estimated FPGA performance for the compositing function

for different frame sizes. When composing the results from four rendering nodes operating in parallel, the estimated speedup ranges from 4× to 4.5×. For eight parallel rendering nodes, the speedup is between 5× and 5.7×. Naturally, the achievable frame rate decreases with an increasing number of subpictures that need to be composed for generating the final picture. However, Figure 6.5 presents only the compositing time. The overall performance of the parallel rendering application will scale with the number of rendering nodes, up to the point where a bottleneck, such as the network saturation bandwidth, is reached.

6.2.2 Implementation

Figure 6.6 shows the architecture of the IMORC based compositing accelerator. The compositing controller is configured with the parameters of the rendering application, such as the number of rendering nodes and the size of the buffers to be composed. These values are translated into parameters for the request core, which encapsulates six IMORC request generator cores.

For the first frame, read requests are sent to the host memory and corresponding write requests to the external memory. For intermediate frames, read requests are sent to the appropriate locations in the host memory and to the compositing buffer in the external memory, as well as corresponding write requests to the compositing buffer in the external

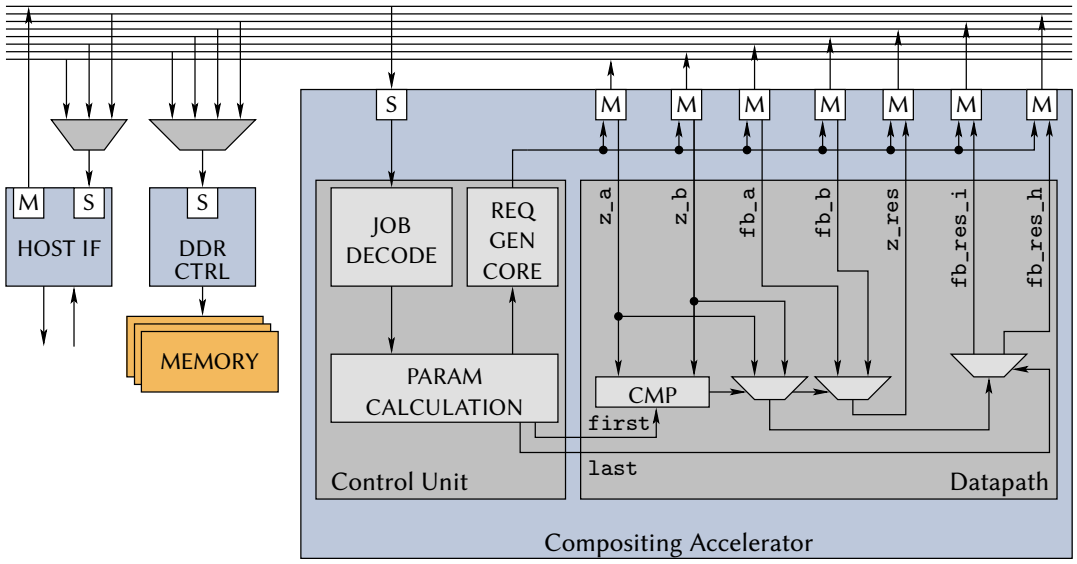


Figure 6.6: Architecture of the compositing accelerator

memory. For the last frame similar read requests are sent, but this time the write requests target the location in the host memory where the frame to be displayed is stored. Requests for frame- and z-buffers are posted seperately, which is the reason for the instantiation of six instead of only three request generator cores.

The datapath of the compositing accelerator in the first step directly forwards data received from the host memory to the compositing buffer in external memory. In the intermediate steps, it compares the z-values received from the host memory to those received from the compositing buffer and uses the result as a select input to two multiplexers. The multiplexers get the two data streams for the frame- and z-buffer received from the host memory and the external memory as input, respectively, and forward the multiplexed values (i. e., those corresponding to the z-buffer with the lower z-value) to the compositing buffer. In the last iteration, only the multiplexed frame buffer is forwarded to the host memory link.

To fully utilize the bandwidth available, the datapath is implemented 64 bit wide, thus operating on two pixels in parallel. Clocked synchronously to the HT interface core at 200 MHz, the datapath's maximum bandwidth is higher than the HT bandwidth. So the accelerator's runtime completely depends on the maximum bandwidth of the HT. The memory controller is configured to 8-cycle bursts and the request size of the request generator cores is set to 128 byte to achieve maximum throughput.

6.2.3 Performance Evaluation

In order to evaluate the overall performance of the parallel renderer, a test system comprising an Intel server connected via Infiniband to the XtremeData XD1000 was implemented (cmp. Figure 6.7). The server contains two Clovertown quad-core processors running at 2.66 GHz and 8 GB of main memory and implements 1–8 rendering nodes. The XD1000 implements the master node including the compositing function. Theoretically, the Infiniband interconnect provides a peak bandwidth of 10 GBit/s. Measurements with the Intel IMB [73] benchmark show that our test setup reaches a sustained Infiniband bandwidth of 700 MB/s.

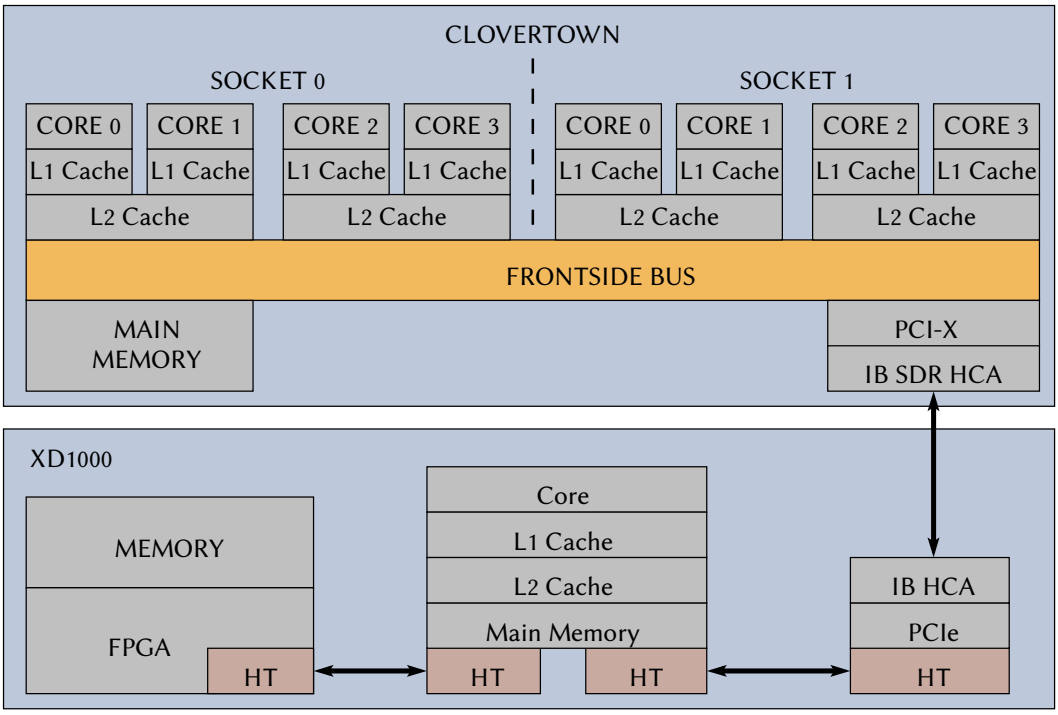


Figure 6.7: Architecture diagram of the test setup

Figure 6.8 compares the overall performance, measured in frames/s, of the purely CPU-based parallel renderer with the FPGA-accelerated parallel renderer. Additionally, the figure shows the frame rates that could be achieved if the Infiniband interconnect would have been fully utilized. Figure 6.8 covers all three system states of the parallel renderer application. The next paragraph discusses these states for a resolution of 1280 × 1024 pixel:

For a small number of rendering nodes (up to three) the performance increases linearly. Since neither the network nor the master node is saturated, there is no benefit from using FPGA acceleration. When the number of rendering nodes increases (four to five nodes), the master becomes a bottleneck. The CPU-based system achieves its maximum frame

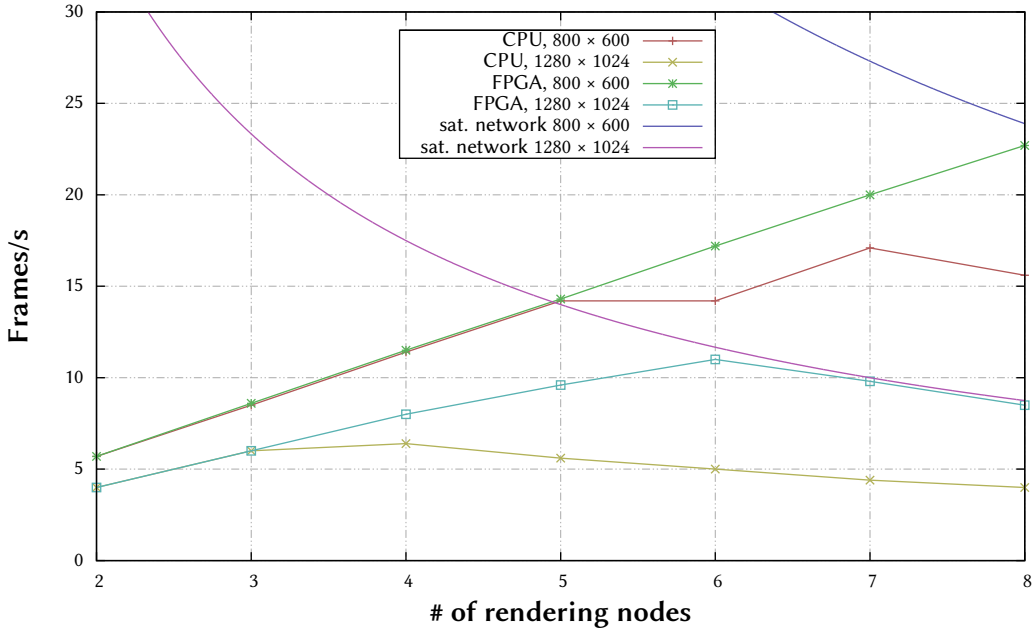


Figure 6.8: Performance values of the complete rendering application

rate for four rendering nodes. In this system state FPGA acceleration is highly useful as the improved compositing performance allows to achieve a higher peak frame rate. For example, for four rendering nodes the performance gain is 1.25×. At a certain point (six or more nodes), the aggregate bandwidth of the rendering nodes saturates the network. Figure 6.8 clearly shows that the performance of the FPGA-accelerated system is limited by the Infiniband bandwidth. Obviously, also in this state FPGA acceleration is beneficial and delivers an improvement in the frame rate of 2.1× for eight rendering nodes.

6.3 K-th Nearest Neighbor Thinning

k-th-nearest-neighbor (KNN) methods are widely used in many areas of science and engineering. In statistics and data analysis, for example, KNN techniques play an important role for the non-parametric estimation of density functions from data samples [83]. Given a set of n data samples, where each sample i is a d-dimensional vector, an Euclidean distance metric σ_i is computed for any pair of samples. For each data sample, the resulting n distance values are sorted in ascending order, i.e., $\sigma_i^1 \leq \sigma_i^2, \ldots, \leq \sigma_i^n$. A KNN density estimate $\hat{f}(i)$ can be formulated by setting: $\hat{f}(i) \propto 1/\sigma_i^k$.

The parameter k is typically chosen as $k \approx \sqrt{n}$ [108]. Thus, the local density around each data sample i is estimated by the reciprocal of the distance to the k-th nearest

neighbor. In other words, a low density means that a d-dimensional sphere with data sample i at its origin has to be rather large in order to contain k data samples.

The KNN approach is also widely applied for solving classifications problems, such as in machine learning, data mining and stochastic optimization [44]. There, a KNN classifier requires a labeled training data set consisting of d-dimensional feature vectors and their class labels. In order to classify a new feature vector, the k nearest training vectors are determined according to some distance metric. Often, a reduction of the size of data samples is desired to reduce both the classification time and the memory required to store the data set. Many reduction techniques fall into the category of condensing or thinning approaches [104] that aim at properly selecting a subset of training vectors from the original data set. Well-known thinning approaches are the condensed nearest neighbor algorithm, the reduced nearest neighbor algorithm, Baram's method, and proximity graph based thinning (see, e.g., [35]).

Recently, some methods related to KNN-based thinning have been successfully accelerated with FPGAs. For example, Yeh et al. present a KNN classifier [129] that operates in the wavelet domain and uses partial distance search to accelerate the classification process. The resulting architecture is integrated as a core with the Altera NIOS CPU softcore. In [42] Chikhi et al. present an FPGA accelerated KNN classifier for content-based image retrieval that achieves a speedup of 45× compared to a software implementation. The related k-means clustering method has also been successfully accelerated in reconfigurable hardware. For example, Saegusa and Maruyama have presented an architecture [102] that can perform k-means clustering on video data in realtime.

Preliminary work to this case study was a hardware accelerator for KNN-based thinning [3] which achieved speedups of one order of magnitude for SPEA2 [130], one of the most popular multi-objective optimizers. SPEA2 optimizes a problem for several typically conflicting objectives by finding reasonable trade-offs between the different objectives. Specifically, SPEA2 approximates the set of Pareto points (Pareto fronts) and relies on KNN methods for thinning out the approximated Pareto fronts. This becomes necessary when the algorithm generates more Pareto points than can be stored in the fixed-size archive, which is rather likely for higher-dimensional problems. Depending on the actual problem being optimized, the KNN thinning technique takes the vast majority of the optimizer's run time.

While the original accelerator presented in [3] achieved good speedups on an embedded Xilinx Virtex-II Pro based FPGA board and on the XtremeData XD1000, its design limited the maximum problem size since the problem and intermediate data were completely stored in on-chip memory. The accelerator presented here overcomes this issue by using the host memory and the external memory when advisable. A previous version of this IMORC based accelerator was presented in [5] and [6]. Further improvements based on the results of these papers is be discussed in the following.

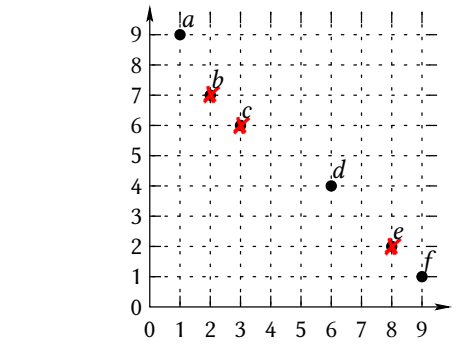
The KNN thinning procedure, shown in Algorithm 6.2, is called with a set $\mathcal{P}$ of different

d-dimensional vectors $p_i = (p_{i1}, p_{i2}, \ldots, p_{id})$ and N, the targeted cardinality of $\mathcal{P}$. It successively eliminates vectors with the shortest Euclidean distance to their neighbors until $\mathcal{P}$ has been reduced to N elements. In each iteration, the algorithm first constructs a distance matrix $\underline{\sigma} = (\sigma_{il})$ from the pair-wise Euclidean distances σ_{il} between all vectors. Sorting $\underline{\sigma}$ row-wise in ascending order defines sorted distance vectors $\sigma_\mathbf{i}'$ of length m. While initially equal to $|\mathcal{P}|$, the number of vectors m is reduced by one in each iteration. Then, the algorithm iterates over all columns of $\underline{\sigma}'$. Starting with column $l = 3$, the rows σ_i' with minimum distance values σ_{il}' among all distances in column l are selected and assigned to set $\mathcal{M}$. If the minimum is unique, the respective row $\sigma_i' \in \underline{\sigma}'$ as well as the corresponding vector p_i are deleted, which reduces the set of vectors $\mathcal{P}$. If the minimum is not unique, the next column of $\underline{\sigma}'$ is considered, which corresponds to checking the distances to the next closest neighbors. If no unique minimal distance is found for all columns of $\underline{\sigma}'$, an arbitrary row having a minimal distance value in the last column is deleted. Obviously, the first two columns never need to be considered since each vector has a distance of 0 to itself (first column) and the distances between pairs of vectors are symmetric (second column).

1: **procedure** KNN_THINNING($\mathcal{P}, N$)
2: **while** $|\mathcal{P}| > N$ **do**
3: compute/update $\sigma_{il} \leftarrow \sqrt{\sum_{j=1}^{d} (p_{ij} - p_{lj})^2}$
4: $\forall$ rows of $\underline{\sigma} : \sigma_\mathbf{i}' \leftarrow sort(\sigma_\mathbf{i})$
5: **for** $l \leftarrow 3, \ldots, m$ **do**
6: $\mathcal{M} \leftarrow \{\sigma_i' \mid \forall \sigma_{jl}' : \sigma_{il}' \leq \sigma_{jl}'\}$
7: **if** $|\mathcal{M}| == 1$ **then**
8: break
9: **end if**
10: **end for**
11: delete arbitrary row $\sigma_i \in \mathcal{M}$ from $\underline{\sigma}'$
12: $\mathcal{P} \leftarrow \mathcal{P} \setminus \{p_i\}$
13: **end while**
14: **end procedure**

Algorithm 6.2: KNN thinning algorithm

The operation of the KNN-based thinning algorithm is illustrated in the example shown in Figure 6.9. The initial population of six 2-dimensional vectors as well as the three vectors that are discarded by three iterations of the thinning algorithm are shown in Figure 6.9(a). The distance matrix $\underline{\sigma}$ for the first iteration is presented in Figure 6.9(b) and the row-wise sorted distance matrix $\underline{\sigma}'$ in Figure 6.9(c). The matrices show the square distances, which is sufficient for this algorithm since the square root operation does not change the ordering



(a)

	a	*b*	*c*	*d*	*e*	*f*
a	0	5	13	50	98	128
b	5	0	2	25	61	85
c	13	2	0	13	41	61
d	50	25	13	0	8	18
e	98	61	41	8	0	2
f	128	85	61	18	2	8

(b)

a	0	5	13	50	98	128
~~*b*~~	~~0~~	~~2~~	~~5~~	~~25~~	~~61~~	~~85~~
c	0	2	13	13	41	61
d	0	8	13	18	25	50
e	0	2	8	41	61	98
f	0	2	18	61	85	128

(c)

a	0	13	50	98	128
c	0	13	13	41	61
d	0	8	13	18	50
~~*e*~~	~~0~~	~~2~~	~~8~~	~~41~~	~~98~~
f	0	2	18	61	128

(d)

a	0	13	50	128
~~*c*~~	~~0~~	~~13~~	~~13~~	~~61~~
d	0	13	18	18
f	0	18	61	61

(e)

Figure 6.9: Example for the KNN-based thinning algorithm in two dimensions: thinning 6 vectors to 3 vectors: (a) shows the original vectors in a graph; (b) shows the original distance table; (c) shows the sorted distance table with a unique minimum found in line *b*, column 3; (d) shows the sorted distance table after *b* was discarded; (e) shows the sorted distance table after *e* was discarded.

of the values. A unique minimum is found in the third column, which leads to the deletion of row *b* from the matrix and vector *b* from the population. In the second iteration, the distance matrix is updated and re-sorted, which results in the matrix of Figure 6.9(d). Again, a unique minimum is identified in the third column and, consequently, row *e* and vector *e* are deleted (Figure 6.9(d)). Finally, the third iteration leads to the deletion of vector *c* (Figure 6.9(e)).

6.3.1 Application Model

Figure 6.10 pictures the modeling flow of the KNN accelerator. In the first step, Algorithm 6.2 is broken up into four major blocks: computing the distance values (line 3), sorting the distance values (line 4), searching for vectors to be discarded (lines 5…10) and discarding them (line 12). The data generation task executed by the overall application starts the KNN controller task that controls the thinning process. It starts the distance calculation task and waits for the discard task to finish. The distance calculator, sort and

search tasks directly start the following task upon finishing. The controller task initiates this procedure until the specified goal is achieved.

In a first optimization step, the repeated execution of the distance calculation and sort task is removed. This is accomplished by storing not only the distances of the vectors in the distance table, but also the IDs of the corresponding vectors. The discard task now not only operates on the original vector table but also on the distance table for removing all references to the vector to be discarded. On the one hand, this increases the size of the distance table and at the same time increases the runtime for all tasks operating on the distance table due to an increased data transfer size. Additionally, the discard task now has to process the complete distance table which further increases its runtime. On the other hand, distance calculation and sorting now are only executed once, thus reducing the overall runtime of the complete task graph.

To better exploit the bandwidth available, in a further optimization the three steps are partitioned into substeps. Each distance calculation task now is responsible for calculating the distances of exactly one vector to each other, hence calculating one row of the distance matrix $\underline{\sigma}$. The same refinement is done for the sorter task, each one sorts one row of $\underline{\sigma}$. Each distance calculation task starts the corresponding sorter task upon finish. However, since the search and discard phase may not be started before all distance calculation tasks and sorter tasks are finished, the sorter tasks are not allowed to directly start the next phase. Instead, an additional control task is inserted to synchronize the finishing of the sorter tasks. The search and distance calculation phase uses two kinds of subtasks, one for searching, the other for discarding. Searching is performed sequentially by executing lines 5–10 of Algorithm 6.2. When the vector to be discarded is found, a number of discard tasks is started, again each one processing one line of $\underline{\sigma}$. An additional discard task is started for updating the vector table $\mathcal{P}$. Control is handed back to the search core, which repeats this procedure until the specified goal is achieved.

6.3.2 IMORC KNN Cores

Given the execution model discussed in the previous section, the next step is to develop a set of cores that the different kinds of tasks can be mapped to. The two storage tasks (vector table and distance table) can be mapped to arbitrary memory locations, the other tasks are to be mapped to custom cores which are discussed in the following paragraphs.

Distance Calculator Core

The Euclidean distance computation task is wrapped into a distance calculator core (cmp. Figure 6.11). The square root function is omitted since squared values are sufficient for comparing distances. The distance calculator core computes one row of the distance table σ_i on each invocation, as specified in the execution model. Multiple distance cal-

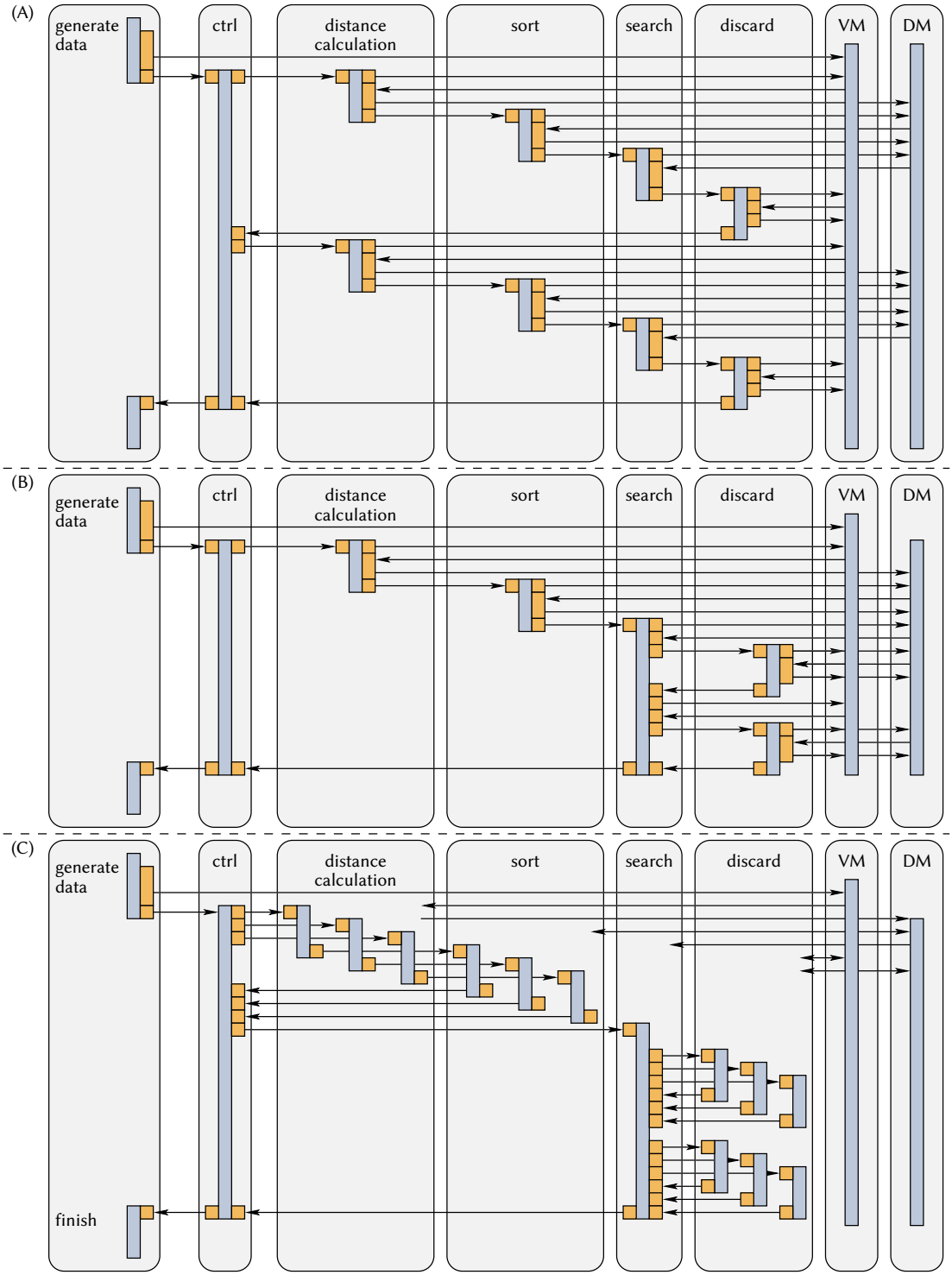


Figure 6.10: Development of the KNN execution model

culation tasks can be mapped to a single distance calculator core, which are processed sequentially.

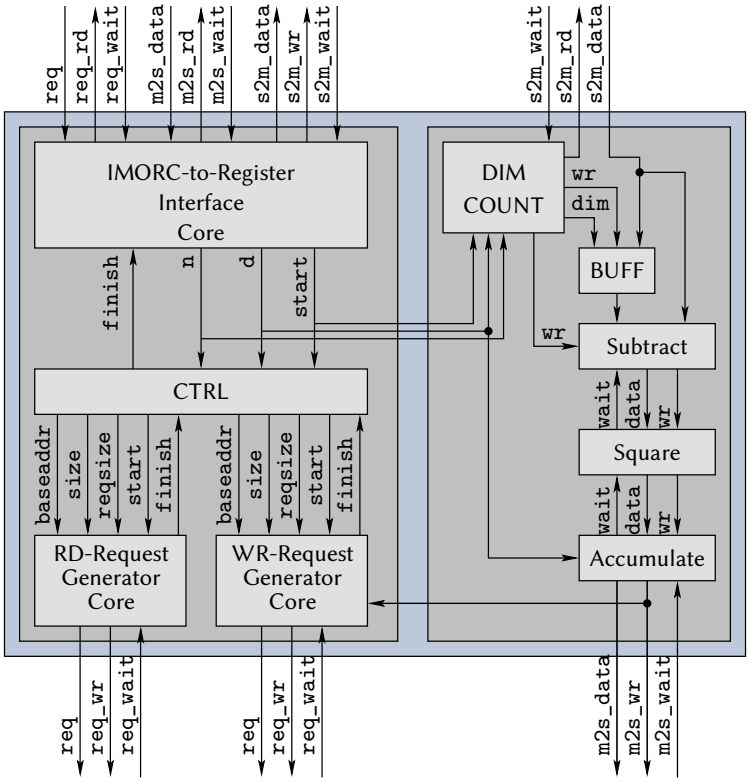


Figure 6.11: Block diagram of the distance calculator

To receive job messages, the distance calculator core contains a slave port, connected to an I2R interface core. A job message comprises the base address of the vectors in the vector memory, the number of vectors and dimensions, m and d, the index i of the vector for which the distances to all vectors have to be determined, and the address in the distance memory where σ_i is to be stored.

Two request generator cores are connected to the I2R interface core, the first one sending read requests to the vector memory, the second one for sending writes to the distance memory. The read request generator is first configured for retrieving the vector i from the vector memory. When finished, it is reconfigured for issuing read requests to fetch all vectors. At the same time, the second request generator is configured for sending write requests to the distance memory for storing the distances.

The datapath reads the first i vector's coordinates from the vector memory's S2M-channel and stores them in an internal buffer. Further coordinates are subtracted from the appropriate coordinates in the buffer, the coordinates distances are squared and added up. The resulting distances are sent to the distance memory's M2S-channel.

When all distance values are written back, the sorter job generator is started, generating a sort job for the computed distances. The I2R interface core is then set IDLE for accepting further distance calculation jobs or state request messages.

Sorter Core

The sort tasks are mapped to a set of cores implementing a variant of bubble sort. On each invocation, the sorter core sorts one row σ_i of the distance table. The job messages are received using a slave port and comprise the number of vectors m and the base address of σ_i in the distance memory. The job message is decoded in an I2R interface core.

Then, the core proceeds as follows: In the first iteration, a request generator is configured for reading n elements of σ_i, i. e., the complete row from distance memory, and for storing the same amount of data back to the same location. The datapath stores the first element received on the S2M channel as the current maximum. If the next element is smaller than or equal to the current maximum, it is directly forwarded to the M2S channel. Otherwise, it becomes the new current maximum and the previous maximum is sent to the M2S channel. At the end of the iteration, the current maximum is written to the M2S channel. Additionally, the position l of the last element in σ_i that has been moved left is remembered. Thus, in the next iteration only l elements of σ_i are to be streamed through the sorter core. Thus, the request generator core is reconfigured for reading and writing l elements. This procedure is repeated until the complete row is sorted. Now the state register of the I2R interface core is reset, so that further sort jobs can be processed or occurring state request messages can be responded to.

Search Core

The search core implementing the search task is invoked with a job message specifying the total number of vectors n, the dimension of the vectors and the goal to achieve (i. e., the number of vectors not to discard). The job is decoded in an I2R interface core. A controller transforms these values into input data for an IMORC request generator, which generates read requests to the elements in the third column of the sorted distance matrix. Since data is stored row-wise, n requests each with a size of 64 bit have to be generated, with an offset of $n \times 8$ byte.

The datapath receives the values from the distance memory and stores the minimum value along with the corresponding vectors' IDs in internal registers. Additionally, the core observes whether this minimum appeared only once in the current data stream (i. e., whether the minimum is a unique minimum). If no unique minimum was found, the controller instructs the request generator to generate requests for the next column. This procedure repeats, until a unique minimum is found or until all columns were searched. In the latter case one of the vectors corresponding to one of the minimums found in the

last column is selected for being discarded.

When finished, one discard job is generated for each row of the distance table. The controller waits until all discard jobs are finished and, if the configured goal is not achieved yet, starts the next search iteration.

Discarder Core

The core implementing the discard tasks is invoked with a job message specifying the total original and current number of vectors n_{orig} and n_{cur}, their dimension d, the id id_{disc} of the vector to discard and the id id_c of the current vector (or row in the distance matrix) to process.

	$(id_c \neq n_{cur}-1) \wedge (id_c \neq id_{disc})$	$id_c = n_{cur}-1$	$id_c = id_{disc}$
rd_base:	$id_c \times n_{orig}$	$id_c \times n_{orig}$	$(n_{cur}-1) \times d$
rd_size:	n_{cur}	n_{cur}	d
wr_base:	$id_c \times n_{orig}$	$id_{discard} \times n_{orig}$	$id_{disc} \times d$
wr_size:	$n_{cur}-1$	$n_{cur}-1$	d

Table 6.2: Parameters of the discarder's request generator

A read-write request generator is configured with the main parameters as defined in Table 6.2. Three different cases exist:

- $(id_c \neq n_{cur}-1) \wedge (id_c \neq id_{disc})$: the requests target the distance memory, reading n_{cur} distances and writing $n_{cur}-1$ distances back into the same row.
- $id_c = n_{cur}-1$: the requests also target the distance memory, reading and writing the same amount of data as in the previous case. This time, the writes do not target the same row as the reads, but the row where the distances origin at vector id_{disc} reside.
- $id_c = id_{disc}$: the requests target the vector memory, reading all coordinates of the last vector ($n_{cur}-1$) and storing them to the location where the coordinates of vector id_{disc} reside.

With these parameters, the vector with the ID id_{disc} is replaced with the last vector in the vector table as well as in the distance table. Since the id of the last vector is now changed, the datapath has to perform this change in the distance table's entries, too. The datapath connected to the distance memory's link reads data from the S2M channel and compares the IDs prepended to the distance value to id_{disc} and to $n_{cur}-1$. Every entry with an ID equal to id_{disc} is discarded, every ID equal to $n_{cur}-1$ is replaced by id_{disc}. Entries not discarded are then written to the M2S channel. The datapath connected to the vector memory's link forwards all values from the S2M channel to the corresponding M2S channel.

Controller Core

The controller core is responsible for sending job requests to the responsible cores. An I2R interface core is used for decoding job messages received from the host. When such a message arrives, the distcalc job generator calculates parameters for n distance calculation jobs and sends them to the distance calculators using an R2I interface core. When all jobs are sent, a read request is sent to the distance calculators in order to receive a notification after all jobs have been finished. Then, the sort waiter is started. Since sort jobs are directly generated and sent to the sorters in the distance calculators, this core only sends a read request to the sorter cores for getting their finishing time. The search job generator sets up parameters for the search module, sends them to the search core using the R2I interface core and waits for completion. Finally, the interrupt generator core is started, which sends an interrupt message to the host link. This way, the host is informed that the current job is finished.

Figure 6.12 shows a block diagram of the controller core. Most elements of the controller core are implemented using the existing IMORC supporting cores. Only a few lines of additional HDL code had to be written for calculating the correct job parameters.

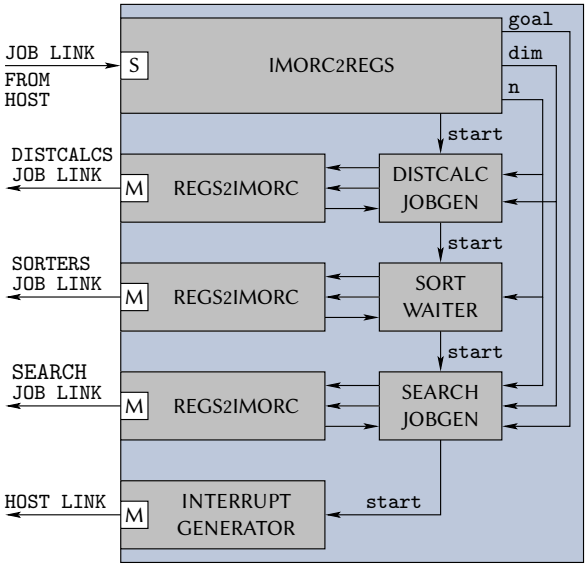


Figure 6.12: Block diagram of the controller core

6.3.3 Architecture Generation

Mapping the application model to the cores presented in the previous section is a straightforward task. The two storage tasks, namely the vector table and the distance table, need

to be mapped to the memory locations on the target platform. The vector table has to be accessed by the host application as well as by the distance calculator and the discard task. The distance table is accessed by all tasks that are implemented in the FPGA. The architecture characterization of the XD1000 presented in the previous chapter states that the CPU can access its own host memory much faster than data on the FPGA, especially when reading data (which is necessary after the thinning procedure is finished). Consequently, the vector table is mapped to the host memory. Since the distance table is only accessed by cores implemented within the FPGA, it can be mapped to the DDR SDRAM, which provides the best performance in this case.

The control and search tasks are directly mapped to one single instance of the control and search core, respectively. To exploit parallelism, the distance calculation, sort and discard tasks can be mapped to multiple cores, which process the jobs in parallel. Hence, additional job distribution tasks have to be introduced, which are implemented by the IMORC farming cores. Figure 6.13 pictures the resulting architecture model of the complete accelerator.

The number of distance calculators, sorters and discarder cores in the accelerator is configurable. All slave arbiters, except that for the sorter job messages, use the default round robin port scheduler. The sort job slave arbiter uses a modified port scheduler, which tries to select one of the links from the discarder cores in round robin manner first and only selects the link from the controller core if all other links are empty. This way, the final status request from the controller is ensured to be posted to the farming core last, when all job messages are already forwarded to the sort compute cores. Figure 6.13 shows the IMORC diagram of the complete accelerator with each two distance calculators, sorters and discarder modules.

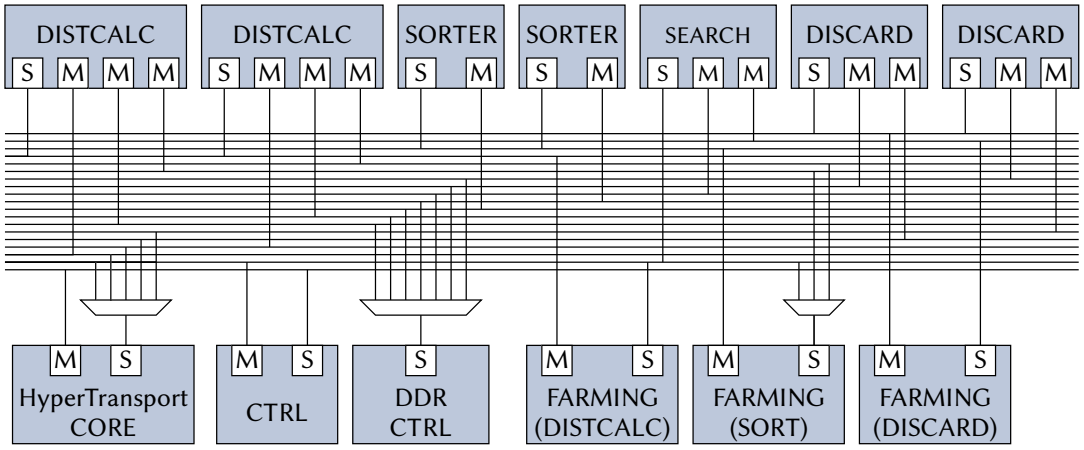


Figure 6.13: Architecture diagram of the accelerator with two distance calculators/sorters/discarders mapped to the XD1000

6.3.4 Numeric Evaluation

Based on the mapping presented in the previous section, several accelerators with different configurations were generated (indicated as ($n_{distcalc} \times n_{sort} \times n_{discard}$)). Table 6.3 demonstrates the resource usage of some basic elements of the accelerators. The last two rows represent the resources used by the complete accelerator and the resources used for the IMORC communication infrastructure of the complete accelerator in the (6 × 6 × 6) configuration. The communication infrastructure hereby takes about 14 % of the overall logic resources.

	ALUTs	REGs	DSP	M512	M4k	M-Ram
EP2S180-3	143520	143520	768	930	768	9
distcalc core	1462	1192	6	-	-	1
sorter core	967	1176	-	-	-	-
search core	1758	1035	4	-	-	-
discarder core	1507	1504	-	-	-	-
CTRL core	782	325	-	-	-	-
IMORC link	99	226	-	-	4	-
bitwidth conv.	68	126	-	2	-	-
farming core	76	29	-	-	-	-
DDR CTRL	1992	2159	-	-	8	-
HT core	8791	5209	-	1	57	1
complete acc.	42084	35492	40	7	383	8
IMORC	5896	4766	-	28	344	-

Table 6.3: Resource requirements for the accelerator

Figure 6.14 shows the speedups these accelerators generate over an optimized software solution for different dimensions and numbers of vectors. The HyperTransport interface core was running at 200 MHz, the DDR SDRAM at 166 MHz. The accelerator clocks were configured to 200 MHz for the CTRL core, 100 MHz for the distance calculator cores, 120 MHz for the sorter cores and 150 MHz for the search and the discard cores.

In all cases the speedup increases with the number of compute cores used. The best configuration presented in the figure is the (6 × 6 × 6)-configuration, which provides speedups of up to 44× over the optimized software solution. The (6 × 1 × 1), (1 × 6 × 1) and (1 × 1 × 6) configurations show that the number of distance calculators used only has a minor impact on the actual computation time. The reason for this is that the distance calculation is only executed once and during execution only one time accesses the complete distance table, while the sorter core and the discarder core need to access the complete distance table several times.

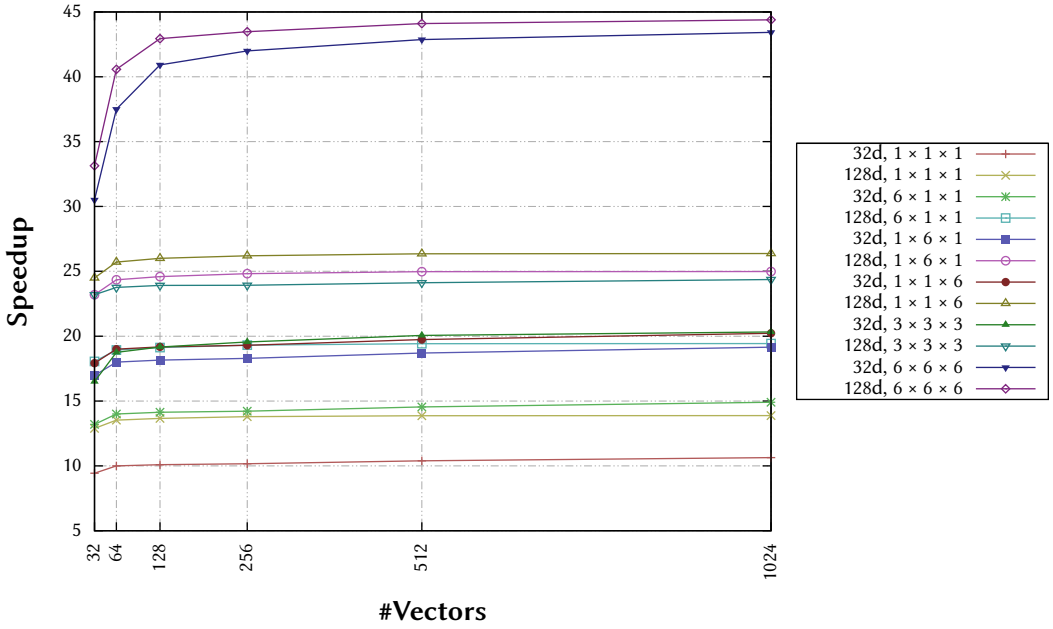


Figure 6.14: Resulting speedups for different accelerator configurations

6.4 Chapter Summary

This chapter presented three case studies with different kinds of applications in order to evaluate the introduced modeling approach and the IMORC architectural template. The first case study, the Cube Cut problem, can intuitively be classified as well-suited for FPGA acceleration. The IMORC modeling approach in this case study supported the analysis of the kernels' communication demand within a real system. In combination with the performance characterization presented in Chapter 5, the modeling approach was useful for finding reasonable parameters for the computation pipeline. The compositing accelerator demonstrates that even algorithms with a low number of computations may benefit from using reconfigurable hardware due to the acceleration gained by an efficient memory access scheme. Supported by the modeling approach and performance characterization a well suited memory mapping and an appropriate access scheme has been identified, which achieves reasonable speedups. Furthermore, the KNN thinning accelerator demonstrates the efficiency of the IMORC architectural template in the case implementing accelerators consisting of several communicating cores. The accelerator uses all features available in the IMORC architectural template. Cores are replicated and the workload is efficiently distributed using the IMORC farming cores. In this way the accelerator can easily be configured for different numbers of compute cores, which simplifies porting the accelerator to other target architectures with a different memory layout and performance.

In all case studies the cores could be implemented with only a few lines of hand written HDL code. The majority of this code is required for generating the cores' datapaths. Control structures are mainly realized by using the IMORC utility cores, such as the request generator cores implementing the data access scheme. Custom code is only required for calculating the input parameters of the request generators and, in the case that cores have to execute multiple iterations, for updating these values with each iteration.

The integrated FIFOs allow the control and datapaths to be implemented independently from each other, simplifying the overall implementation effort. The integrated bitwidth conversion modules further simplified the core development since cores can always access data at their native bitwidth, regardless of the actual target memory resource. Due to the slave arbitration combined with the integrated FIFOs, memories can be accessed at their maximum performance even when single cores are not able to process data at the same rate.

During development of the accelerators, the performance counters have provided the basis for identifying performance bottlenecks and for further system optimization. Additionally, the counters have supported the bug analysis of the accelerator, especially in the kNN application. While calculation faults typically cannot be identified by these counters, communication errors can be related to individual cores. If, for example, a core's datapath expects less data to be processed than requested by the corresponding request generator, the corresponding data FIFO will never become empty again.

Concluding, the case studies demonstrate the benefits of the IMORC development flow and of the IMORC architectural template for the development of reconfigurable accelerators. IMORC simplifies accelerator development in many aspects — creating the complete accelerators from scratch each time would have taken much more time.

7 Conclusion and Outlook

This chapter summarizes the contributions of this thesis, draws a conclusion and gives an outlook on future directions.

7.1 Contributions

This thesis adds the following contributions to the state of the art in reconfigurable high-performance computing:

- It introduces a novel modeling flow for reconfigurable accelerators. The model is flexible in regard to the level of expressible detail. It consists of an architecture and an execution model which are generated in parallel and are useful for estimating the suitability of reconfigurable accelerators before implementation. This enables to identify bottlenecks already during the design phase.

- An architectural template for generating reconfigurable accelerators is presented that was designed to support implementing accelerators as specified by the modeling technique presented. The architectural template includes a communication infrastructure which enables cores to communicate with each other at high speed with only little impact caused by contention. Supporting cores for functionalities often needed are provided for further speedup of the design process. Additionally, performance monitoring methods are introduced for identifying bottlenecks in the running system.

- An architecture characterization framework is presented that enables an accurate analysis of the target platform. The bandwidth achievable when communicating to the different resources of the target platform can be measured with different

communication schemes. Such information is necessary for implementing well performing accelerators.

7.2 Conclusion

In this thesis, a method for generating reconfigurable accelerators for high-performance computing was introduced. The IMORC modeling approach supports the developer in several stages of the design process. First, it forms a decision basis whether reconfigurable computing makes sense for a given application or algorithm. Second, it helps in designing an efficient architecture implementing the desired algorithms. And third, it may be used as a basis for an initial performance analysis before actually implementing the accelerator.

The IMORC architectural template disburdens the designer by providing integrated functionalities that are often needed by reconfigurable accelerators. It provides an efficient infrastructure for connecting cores within a FPGA to an integrated system. The FIFOs inserted into the IMORC links decouple the datapath of cores from the control logic, enabling a straight-forward core design. The bitwidth conversion modules additionally make memory resources transparently accessible. The slave arbitration in combination with the integrated FIFOs enable a maximum utilization of the different memory resources even if single cores are not able to fully exploit the bandwidth provided. Utility cores further reduce development time, enabling cores to be implementable with only few lines of custom HDL code.

The benefits of the IMORC modeling and implementation approach were demonstrated in several case studies. Especially the compositing accelerator case study has shown that even applications with very few computations, which intuitively should not perform well on reconfigurable architectures, may benefit from FPGA acceleration due to an efficient utilization of available memory resources. In all case studies, the IMORC approach greatly supported the generation of the accelerators. The modeling approach led to an efficient architecture that could be mapped to the FPGA in a straight-forward manner. The distinction between request and data in the IMORC links combined with the FIFO storage enabled to implement control functions and datapaths independently from each other. The control structures could in most cases directly be implemented using the different variants of request generator cores, with only a small amount of additional code used for calculating the appropriate parameters. Datapaths are implemented as a computation pipeline in all cases.

During implementation, the integrated load sensors significantly supported to identify performance bottlenecks and the further optimization of the accelerators.

7.3 Future Directions

Future work might incorporate the following interesting directions:

- While the IMORC architecture is implemented with a focus on portability, currently only the XtremeData XD1000 system is fully supported. While the case studies presented were also synthesized for current Xilinx devices to prove the vendor neutrality of the infrastructure and supporting cores, currently no host interfaces and external memory controllers for further systems are provided. Recently, several Nallatech systems with a Xilinx Virtex V FPGA attached to the Intel FSB became available at the University of Paderborn, which are an interesting target for IMORC. Another interesting target architecture recently installed at the PC^2 is the Convey HC1, which provides four user programmable Xilinx Virtex V FPGAs. Additionally, the system provides a large amount of DDR2 SDRAM, which is coherently mapped into the host CPU's address space and can be accessed by the FPGAs at a peak rate of up to 80 GB/s.

- The IMORC modeling approach currently supports developers in estimating the suitability of reconfigurable computing for high-performance computing and in the design phase of the accelerator. Currently, it does not calculate concrete performance values but only performs a rough estimation of the benefits reconfigurable hardware implies on given algorithms. Possible future work might focus on the suitability of such modeling approaches for gathering more concrete performance values. Additionally, the modeling approach could be evaluated targeting different kinds of hardware, such as GPUs or vector processors like ClearSpeed.

- The current IMORC architectural template primarily consists of modules implemented in VHDL that are customizable using VHDL generics and are to be instantiated in the application design. Currently, a basic code generator exists that can simplify the task of generating the architecture. Further work could include a more versatile architecture generation tool, for example including a graphical design environment or a HLL datapath compiler.

Acronyms

IMORC Acronyms

REQ	IMORC Request Channel
M2S	IMORC Master-to-Slave Channel
S2M	IMORC Slave-to-Master Channel
I2R	IMORC-to-Register Interface Core
R2I	Register-to-IMORC Interface Core

Further Acronyms

BSP	Bulk Synchronous Programming
CPU	Central Processing Unit
CRS	Compressed Row Storage
DSP	Digital Signal Processor
FPGA	Field Programmable Gate Array
FSB	Front Side Bus
GPU	Graphics Processing Unit
HLL	High-Level Language
HPC	High-Performance Computing
HPRC	High-Performance Reconfigurable Computing
IB	Infiniband
IMB	Intel MPI Benchmark
IR	Intermediate Representation
JTAG	Joint Test Action Group
LLVM	Low Level Virtual Machine
LogP	Latency, overhead, gap and Processors

MIMD	Multiple Instruction stream, Multiple Data stream
MPI	Message Passing Interface
MVP	Mitrion Virtual Processor
NUMA	Non-Uniform Memory Access
OPB	On-chip Peripheral Bus
OSCI	Open SystemC Initiative
PC^2	Paderborn Center for Parallel Computing
PCB	Printed Circuit Board
PRAM	Parallel Random Access Machine
QSM	Queuing Shared Memory
RAM	Random Access Machine
RASP	Random Access Stored Program
PLB	Processor Local Bus
RTL	Register Transfer Level
SMP	Symetric Multiprocessing
SOPC	System On Programmable Chip
UPB	University of Paderborn
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circurit

List of Figures

List of Tables

Author's Publications

[1] Marco Platzner, Sven Döhre, Markus Happe, Tobias Kenter, , Ulf Lorenz, Tobias Schumacher, Andre Send, and Alexander Warkentin. The GOmputer: Accelerating GO with FPGAs. In *Proc. Int. Conf. on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, pages 245–251. CSREA Press, 2008.

[2] Tobias Schumacher, Enno Lübbers, Paul Kaufmann, and Marco Platzner. Accelerating the Cube Cut Problem with an FPGA-augmented Compute Cluster. In *Proceedings of the ParaFPGA Symposium, International Conference on Parallel Computing (ParCo)*, Aachen/Jülich, Germany, Sep. 2007.

[3] Tobias Schumacher, Robert Meiche, Paul Kaufmann, Enno Lübbers, Christian Plessl, and Marco Platzner. A Hardware Accelerator for k-th Nearest Neighbor Thinning. In *Proc. Int. Conf. on Engineering of Reconfigurable Systems and Algorithms (ERSA)*, pages 245–251. CSREA Press, 2008.

[4] Tobias Schumacher, Christian Plessl, and Marco Platzner. IMORC: An infrastructure for performance monitoring and optimization of reconfigurable computers. Many-core and Reconfigurable Supercomputing Conference (MRSC), April 2008.

[5] Tobias Schumacher, Christian Plessl, and Marco Platzner. An Accelerator for k-th Nearest Neighbor Thinning Based on the IMORC Infrastructure. In *Proc. Int. Conf. on Field Programmable Logic and Applications (FPL)*, pages 338–344. IEEE, September 2009. (**Nominated for best paper award**).

[6] Tobias Schumacher, Christian Plessl, and Marco Platzner. IMORC: Application Mapping, Monitoring and Optimization for High-Performance Reconfigurable Computing. In *Proc. IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM)*. IEEE Computer Society, April 2009. (**won a HiPEAC publication award**).

[7] Tobias Schumacher, Tim Süß, Christian Plessl, and Marco Platzner. Communication Performance Characterization for Reconfigurable Accelerator Design on the XD1000. In *Proc. Int. Conf. on ReConFigurable Computing and FPGAs (ReConFig)*. Conference Publishing Services (CPS), 2009.

[8] Tobias Schumacher, Tim Süß, Christian Plessl, and Marco Platzner. FPGA Acceleration of Communication-bound Streaming Applications: Architecture Modeling and a 3D Image Compositing Case Study. *International Journal of Reconfigurable Computing (IJRC)*, vol. 2011, 2011. Article ID 760954.

Bibliography

[9] Advanced Risc Machines (ARM). Website, 2011. http://www.arm.com.

[10] ClearSpeed. Website, 2011. http://www.clearspeed.com.

[11] Cray. Website, 2011. http://www.cray.com.

[12] Edinburgh Parallel Computing Centre (EPCC). Website, 2011. http://www.epcc.ed.ac.uk.

[13] FPGA High Performance Computing Alliance (FHPCA). Website, 2011. http://www.fhpca.org.

[14] HyperTransport Consortium. Website, 2011. http://www.hypertransport.org.

[15] Nallatech. Website, 2011. http://www.nallatech.com.

[16] Netperf networking benchmark. Website, 2011. http://www.netperf.org/.

[17] NSF Center for High-Performance Reconfigurable Computing (CHREC). Website, 2011. http://www.chrec.org.

[18] OpenCores. Website, 2011. http://www.opencores.org/.

[19] OpenFabrics. Website, 2011. http://www.openfabrics.org/.

[20] OpenMPI. Website, 2011. http://www.open-mpi.org/.

[21] OProfile. Website, 2011. http://oprofile.sourceforge.net.

[22] RAMspeed benchmark. Website, 2011. http://www.alasir.com/software/ramspeed.

[23] Silicon Graphics (SGI). Website, 2011. http://www.sgi.com.

[24] SRC. Website, 2011. http://www.srccomp.com.

[25] Top500 supercomputer sites. Website, 2011. http://www.top500.org/.

[26] Valgrind. Website, 2011. http://valgrind.org.

[27] Aeroflex Gaisler. GRLIB IP Library. Website, 2011. http://www.gaisler.com/products/grlib/grlib.html.

[28] L. Agarwal, M. Wazlowski, and S. Ghosh. An asynchronous approach to efficient execution of programs on adaptive architectures utilizing fpgas. In *FPGAs for Custom Computing Machines, 1994. Proceedings. IEEE Workshop on*, pages 101 –110, apr 1994.

[29] L. Agarwal, M. Wazlowski, and S. Ghosh. An asynchronous approach to synthesizing custom architectures for efficient execution of programs on fpgas. In *Parallel Processing, 1994. ICPP 1994. International Conference on*, volume 2, pages 290 –294, aug. 1994.

[30] Albert Alexandrov, Mihai F. Ionescu, Klaus E. Schauser, and Chris Scheiman. LogGP: incorporating long messages into the LogP model—one step closer towards a realistic model for parallel computation. In *Proceedings of the seventh annual ACM symposium on Parallel algorithms and architectures*, SPAA '95, pages 95–105, New York, NY, USA, 1995. ACM.

[31] Altera. *Stratix II Device Handbook*, 2009.

[32] Altera. DSP Builder. Website, 2011. http://www.altera.com/products/software/products/dsp/dsp-builder.html.

[33] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS '67 (Spring): Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485, New York, NY, USA, 1967. ACM.

[34] Don Anderson and Jay Trodden. *HyperTransport System Architecture*. Addison Wesley, February 2003.

[35] F. Bajramovic, F. Mattern, Nicholas Butko, and J. Denzler. A Comparison of Nearest Neighbor Search Algorithms for Generic Object Recognition. In *Proc. of Advanced Concepts for Intelligent Vision Systems (ACIVS)*, pages 1186–1197. Springer, 2006.

[36] Egon Balas and Robert Jeroslow. Canonical cuts on the unit hypercube. *SIAM Journal on Applied Mathematics*, 23(1):61–69, 1972.

[37] P. Banerjee, D. Bagchi, M. Haldar, A. Nayak, V. Kim, and R. Uribe. Automatic conversion of floating point matlab programs into fixed point fpga based hardware design. In *Proceedings of the Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, volume 0, page 263, Los Alamitos, CA, USA, 2003. IEEE Computer Society.

[38] V. Berman. Standards: The p1685 ip-xact ip metadata standard. *Design Test of Computers, IEEE*, 23(4):316 –317, apr. 2006.

[39] Kiran Bondalapati and Viktor K. Prasanna. Dynamic Precision Management for Loop Computations on ReconfigurableArchitectures. In *Proceedings of the Seventh Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, FCCM '99, pages 249–258, Washington, DC, USA, 1999. IEEE Computer Society.

[40] João M.P. Cardoso and Pedro C. Diniz. *Compilation Techniques for Reconfigurable Architectures*. Springer, 2009.

[41] M.L. Chang and S. Hauck. Precis: a design-time precision analysis tool. In *Proceedings of the 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 2002*, pages 229–238, 2002.

[42] Rayan Chikhi, Steven Derrien, Auguste Noumsi, and Patrice Quinton. Combining flash memory and FPGAs to efficiently implement a massively parallel algorithm for content-based image retrival. In *Proc. Int. Workshop on Reconfigurable Computing: Architectures, Tools and Applications*, number 4419 in LNCS, pages 247–258, 2007.

[43] Stephen A. Cook and Robert A. Reckhow. Time-bounded random access machines. In *STOC '72: Proceedings of the fourth annual ACM symposium on Theory of computing*, pages 73–80, New York, NY, USA, 1972. ACM.

[44] T.M. Cover and P.E. Hart. Nearest Neighbor Pattern Classification. *IEEE Transactions on Information Theory*, 13:21–27, 1967.

[45] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. Logp: towards a realistic model of parallel computation. *ACM SIGPLAN Notices*, 28(7):1–12, 1993.

[46] Clifford E. Cummings and Peter Alfke. Simulation and synthesis techniques for asynchronous fifo design with asynchronous pointer comparisons. In *Proceedings of the Synopsys Users Group (SNUG), San Jose, CA*, 2002.

[47] John Curreri, Seth Koehler, Alan D. George, Brian Holland, and Rafael Garcia. Performance analysis framework for high-level language applications in reconfigurable computing. *ACM Trans. Reconfigurable Technol. Syst.*, 3(1):1–23, 2010.

[48] John Curreri, Seth Koehler, Brian Holland, and Alan D. George. Performance analysis with high-level languages for high-performance reconfigurable computing. In *Proc. IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM)*. IEEE Computer Society, April 2008.

[49] Giovanni De Micheli. Hardware synthesis from c/c++ models. In *DATE '99: Proceedings of the conference on Design, automation and test in Europe*, page 80, New York, NY, USA, 1999. ACM.

[50] Pedro Diniz, Mary Hall, Joonseok Park, Byoungro So, and Heidi Ziegler. Automatic mapping of c to fpgas with the defacto compilation and synthesis system. *Microprocessors and Microsystems*, 29(2-3):51 – 62, 2005. Special Issue on FPGA Tools and Techniques.

[51] Chrilly Donninger and Ulf Lorenz. The chess monster hydra. In Jürgen Becker, Marco Platzner, and Serge Vernalde, editors, *Field Programmable Logic and Application*, volume 3203 of *Lecture Notes in Computer Science*, pages 927–932. Springer Berlin / Heidelberg, 2004. 10.1007/978-3-540-30117-2_101.

[52] Calvin C. Elgot and Abraham Robinson. Random-access stored-program machines, an approach to programming languages. *Journal of the ACM*, 11(4):365–399, 1964.

[53] M.R. Emamy-Khansary. On the cuts and cut number of the 4-cube. In *Journal of Combinatorial Theory Series A 41*, pages 211–227, 1986.

[54] FFTW. Fast Fourier Transform Benchmarks. Website, 2011. `http://www.fftw.org/`.

[55] Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21(9):948 –960, 9 1972.

[56] Steven Fortune and James Wyllie. Parallelism in random access machines. In *STOC '78: Proceedings of the tenth annual ACM symposium on Theory of computing*, pages 114–118, New York, NY, USA, 1978. ACM.

[57] Free Software Foundation. GNU Binutils. Website, 2011. `http://www.gnu.org/software/binutils`.

[58] A.A. Gaffar, O. Mencer, W. Luk, P.Y.K. Cheung, and N. Shirazi. Floating-point bitwidth analysis via automatic differentiation. In *Proceedings of the 2002 IEEE International Conference on Field-Programmable Technology, 2002. (FPT).*, pages 158–165, Dec. 2002.

[59] Gilles Kahn. The Semantics of a Simple Language for Parallel Programming, Information Processing. In *Proceedings of the IFIP Congress*, pages 471–475. North-Holland Publishing Company, 1974.

[60] Maya B. Gokhale and Janice M. Stone. Napa c: Compiling for a hybrid risc/fpga architecture. In *Proceedings of the 1998 Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '98)*, page 126, Los Alamitos, CA, USA, 1998. IEEE Computer Society.

[61] Maya B. Gokhale, Janice M. Stone, Jeff Arnold, and Mirek Kalinowski. Stream-oriented fpga computing in the streams-c high level language. In *Proceedings of the 2000 Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '00)*, page 49, Washington, DC, USA, 2000. IEEE Computer Society.

[62] Zhi Guo, Betul Buyukkurt, and Walid Najjar. Input data reuse in compiling window operations onto reconfigurable hardware. *SIGPLAN Not.*, 39(7):249–256, 2004.

[63] András Hajnal, Wolfgang Maass, Pavel Pudlák, György Turán, and Márió Szegedy. Threshold circuits of bounded depth. *J. Comput. Syst. Sci.*, 46(2):129–154, 1993.

[64] Jeff Hammes, Robert Rinker, Wim Böhm, Walid Najjar, Bruce Draper, Ross Beveridge, and Bruce Draper Ross Beveridge. Cameron: High level language compilation for reconfigurable systems. In *International Conference on Parallel Architectures and Compilation Techniques (PACT*, pages 236–244, 1999.

[65] Brian Holland, Allan D. George, and Herman Lam. Integrating application specification and performance prediction for strategic design-space exploration. In *Proceedings of the Int. Conf. on Engineering of Reconfigurable Systems and Algorithms (ERSA '10)*, 2010.

[66] Brian Holland, Karthik Nagarajan, Chris Conger, Adam Jacobs, and Alan D. George. Rat: A methodology for predicting performance in application design migration to fpgas. In *Proceedings of High-Performance Reconfigurable Computing Technologies and Applications Workshop (HPRTCA)*, 2007.

[67] Brian Holland, Karthik Nagarajan, and Alan D. George. Rat: Rc amenability test for rapid performance prediction. *ACM Trans. Reconfigurable Technol. Syst.*, 1(4):1–31, 2009.

[68] Brian Holland, M. Vacas, V. Aggarwal, R. DeVille, I. Troxel, and A. George. Survey of c-based application mapping tools for reconfigurable computing. In *Proceedings of the 8th International Conference on Military and Aerospace Programmable Logic Devices (MAPLD '04)*, Washington, DC, USA, September 2005.

[69] HyperTransport Technology Consortium. *HyperTransport I/O Link Specification.*

[70] IBM. CoreConnect Bus Architecture. Website, 2011. `https://www-01.ibm.com/chips/techlib/techlib.nsf/products/CoreConnect_Bus_Architecture`.

[71] Imperial College London. The AXEL Project – power of heterogeneous cluster. Website, 2010. `http://cc.doc.ic.ac.uk/projects/prj_axel/`.

[72] Impulse Accelerated Technologies. ImpulseC. Website, 2011. `http://www.impulseaccelerated.com`.

[73] Intel. Intel mpi benchmarks. Website, 2011. `http://software.intel.com/en-us/articles/intel-mpi-benchmarks`.

[74] Intel. VTune. Website, 2011. `http://software.intel.com/en-us/intel-vtune`.

[75] JeffHammes, Robert Rinker, Wim Böhm, and Walid Najjar. Compiling a high-level language to reconfigurable systems. In *Compiler and Architecture Support for Embedded Systems (CASES)*, 1999.

[76] John D. McCalpin. STREAM: Sustainable Memory Bandwidth in High Performance Computers. Technical report, University of Virginia, Charlottesville, Virginia, 1991-2007. A continually updated technical report. http://www.cs.virginia.edu/stream/.

[77] John D. McCalpin. Memory Bandwidth and Machine Balance in Current High Performance Computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, December 1995.

[78] Alexander Kaganov, Paul Chow, and Asif Lakhany. Fpga acceleration of monte-carlo based credit derivative pricing. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL'08)*, pages 329–334, 2008.

[79] Tobias Kenter, Marco Platzner, Christian Plessl, and Michael Kauschke. Performance estimation for the exploration of CPU-accelerator architectures. In Omar Hammami and Sandra Larrabee, editors, *Proc. Workshop on Architectural Research Prototyping (WARP)*, June 2010.

[80] Seth Koehler, John Curreri, and Alan D. George. Performance analysis challenges and framework for high- performance reconfigurable computing. *Parallel Computing*, 34(4-5):217–230, 2008.

[81] David Ku and Giovanni DeMicheli. Hardwarec – a language for hardware design (version 2.0). Technical report, Stanford University, Stanford, CA, USA, 1990.

[82] R. Lipsett, E. Marschner, and M. Shahdad. Vhdl - the language. *Design Test of Computers, IEEE*, 3(2):28 –41, apr. 1986.

[83] D.G. Loftsgarden and C.P. Quesenberry. A Nonparametric Estimate of a Multivariate Density Function. *The Annals of Mathematical Statistics*, 31:1049–1051, 1965.

[84] Mentor Graphics. Catapult C. Website, 2011. `http://www.mentor.com/esl/catapult/overview`.

[85] Mentor Graphics. Handel-C. Website, 2011. `http://www.mentor.com/products/fpga/handel-c/`.

[86] MentorGraphics. HDL Designer. Websiter, 2011. `http://www.mentor.com/products/fpga/hdl_design/hdl_designer_series`.

[87] Mitrionics. Mitrion-C. Website, 2011. `http://www.mitrionics.com`.

[88] Steve Molnar, Michael Cox, David Ellsworth, and Henry Fuchs. A Sorting Classification of Parallel Rendering. *IEEE Comput. Graph. Appl.*, 14:23–32, July 1994.

[89] Nallatech. Dime-C. Website, 2011. `http://www.nallatech.com/Development-Tools/dime-c.html`.

[90] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM SIGPLAN Notices*, 42(6):89–100, 2007.

[91] Netlib Repository. High-Performance Linpack. Website, 2011. `http://www.netlib.org/benchmark/hpl/`.

[92] Netlib Repository. Sparse Matrix Benchmarks. Website, 2011. `http://www.netlib.org/benchmark/sparsebench/`.

[93] Novikoff. On convergence proofs on perceptrons. In *Proceedings of the Symposium on the Mathematical Theory of Automata*, volume 12, pages 615–622. Polytechnic Institute of Brooklyn, 1962.

[94] P. O'Neil. Hyperplane cuts of an *n*-cube. In *Discrete Mathematics 1*, pages 193–195, 1971.

[95] Open SystemC Initiative. SystemC. Website, 2011. `http://www.systemc.org/home/`.

[96] Open Verilog International (OVI). *OVI Verilog HDL Language Reference Manual*, 1991.

[97] Opencores. *WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores*.

[98] Iyad Ouaiss, Sriram Govindarajan, Vinoo Srinivasan, Meenakshi Kaul, and Ranga Vemuri. An Integrated Partitioning and Synthesis System for Dynamically Reconfigurable Multi-FPGA Architectures. In *In Proceedings of Parallel and Distributed Processing*, pages 31–36. Springer, 1998.

[99] Paderborn Center for Parallel Computing. PC^2 Benchmark Center. Website, 2011. `http://pc2.uni-paderborn.de/people/jens-simon/benchmarkcenter/`.

[100] Vijaya Ramachandran. Qsm: A general purpose shared-memory model for parallel computation. In *Proceedings of the 17th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'97)*, pages 1–5, 1997.

[101] Casey Reardon, Brian Holland, Alan D. George, Greg Stitt, and Herman Lam. Rcml: An environment for estimation modeling of reconfigurable computing systems. *ACM Transactions on Embedded Computing Systems (TECS), to appear*, 2010.

[102] Takashi Saegusa and Tsutomu Maruyama. An FPGA implementation of k-means clustering for color images based on Kd-tree. *Proc. Int. Conf. on Field Programmable Logic and Applications (FPL)*, pages 1–6, August 2006.

[103] Michael E. Saks. Slicing the hypercube. In Keith Walker, editor, *Surveys in Combinatorics*, pages 211–255. Cambridge University Press, 1993.

[104] J.S. Sanchez, J.M. Sotoca, and F. Pla. Efficient Nearest Neighbor Classification with Data Reduction and Fast Search Algorithms. In *Proc. IEEE Int. Conf. on Systems, Man and Cybernetis*, pages 4757–4762. IEEE, 204.

[105] Andrew G. Schmidt and Ron Sass. Quantifying effective memory bandwidth of platform fpgas. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '07)*, pages 337–338. IEEE Computer Society, 2007.

[106] Lesley Shannon and Paul Chow. Simplifying the integration of processing elements in computing systems using a programmable controller. In *FCCM '05: Proceedings of the 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 63–72, Washington, DC, USA, 2005. IEEE Computer Society.

[107] Lesley Shannon and Paul Chow. Simppl: an adaptable soc framework using a programmable controller ip interface to facilitate design reuse. *IEEE Trans. Very Large Scale Integr. Syst.*, 15(4):377–390, 2007.

[108] Bernard W. Silverman. *Density Estimation for Statistics and Data Analysis*. Chapman & Hall/CRC, April 1986.

[109] J. Simon and J.-M. Wierum. The latency-of-data-access model for analyzing parallel computation. *Information Processing Letters*, 66(5):255–261, 1998.

[110] David Slogsnat, Alexander Giese, and Ulrich Brüning. A versatile, low latency HyperTransport core. In *Proc. Int. Symp. on Field Programmable Gate Arrays (FPGA)*, pages 45–52, New York, NY, USA, 2007. ACM.

[111] Melissa C. Smith. *Analytical modeling of high performance reconfigurable computers: prediction and analysis of system performance*. PhD thesis, University of Tennessee, 2003. Major Professor-Gregory D. Peterson.

[112] Melissa C. Smith and Gregory D. Peterson. Analytical modeling for high performance reconfigurable computers. In *Proceedings of the International Symposium on Performance Evaluation of Computer and Telecommunication Systems (SPECTS'02), San Diego, California*, July 2002.

[113] Melissa C. Smith and Gregory D. Peterson. Parallel application performance on shared high performance reconfigurable computing resources. *Performance modelling and evaluation of high-performance parallel and distributed system*, 60(1-4):107–125, 2005.

[114] C. Sohler and M. Ziegler. Computing Cut Numbers. In *Proceedings of the 12th Annual Canadian Conference on Computational Geometry (CCCG 2000)*, pages 73–79, 2000.

[115] S.A. Spacey, W. Luk, P.H.J. Kelly, and D. Kuhn. Rapid design space visualisation through hardware/software partitioning. In *Proceedings of the 5th Southern Conference on Programmable Logic (SPL '09)*, pages 159 –164, apr. 2009.

[116] Simon A. Spacey. 3s: Program instrumentation and characterization framework. Technical report, Imperial College London, Department of Computing, 2008.

[117] SRC. Carte Programming Environment. Website, 2011. `http://www.srccomp.com/techpubs/carte.asp`.

[118] Craig Steffen. Parametrization of algorithms and fpga accelerators to predict performance. In *Proceedings of the Reconfigurable System Summer Institute (RSSI)*, pages 17–20, 2007.

[119] Sun Microsystems. Sun Studio Performance Tools. Website, 2009. `http://developers.sun.com/solaris/articles/perftools.html`.

[120] Synopsys. Synphony Model Compiler. Website, 2011. http://www.synopsys.com/Systems/BlockDesign/HLS/Pages/Synphony-Model-Compiler.aspx.

[121] The MathWorks. Matlab/Simulink. Website, 2011. http://www.mathworks.com.

[122] Justin Tripp, Preston Jackson, and Brad Hutchings. Sea cucumber: A synthesizing compiler for fpgas. In *Field-Programmable Logic and Applications: Reconfigurable Computing Is Going Mainstream*, pages 51–72. Springer, 2002.

[123] Kuen Hung Tsoi and Wayne Luk. Axel: a heterogeneous cluster with fpgas and gpus. In *FPGA '10: Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays*, pages 115–124, New York, NY, USA, 2010. ACM.

[124] University of Paderborn. The Cut Number Home Page. http://wwwcs.uni-paderborn.de/cs/ag-madh/WWW/CUBECUTS, 2011.

[125] Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990.

[126] Guiming Wu, Yong Dou, Yuanwu Lei, Jie Zhou, Miao Wang, and Jingfei Jiang. A fine-grained pipelined implementation of the linpack benchmark on fpgas. In *Proceedings of the Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '09)*, volume 0, pages 183–190, Los Alamitos, CA, USA, 2009. IEEE Computer Society.

[127] Xilinx, Inc. System Generator for DSP. Website, 2011. http://www.xilinx.com/tools/sysgen.htm.

[128] XtremeData, Inc., Schaumburg, IL, USA. *XD1000 Development System*, 2008.

[129] Yao-Jung Yeh, Hui-Ya Li, Wen-Jyi Hwang, and Chiung-Yao Fang. FPGA implementation of kNN classifier based on wavelet transform and partial distance search. In *Proc. Scandinavian Conf. on Image Analysis (SCIA)*, number 4522 in LNCS, pages 512–521. Springer-Verlag, 2007.

[130] Eckart Zitzler, Marco Laumanns, and Lothar Thiele. SPEA2: Improving the strength pareto evolutionary algorithm for multiobjective optimization. In *Evolutionary Methods for Design, Optimisation and Control with Application to Industrial Problems (EUROGEN 2001)*, pages 95–100. International Center for Numerical Methods in Engineering (CIMNE), 2002.